VISIT…

GRAPH THEORY, COMBINATORICS AND ALGORITHMS

INTERDISCIPLINARY APPLICATIONS

GRAPH THEORY, COMBINATORICS AND ALGORITHMS

INTERDISCIPLINARY APPLICATIONS

Edited by

Martin Charles Golumbic
Irith Ben-Arroyo Hartman

Springer

Martin Charles Golumbic
University of Haifa, Israel

Irith Ben-Arroyo Hartman
University of Haifa, Israel

Library of Congress Cataloging-in-Publication Data

Graph theory, combinatorics, and algorithms / [edited] by Martin Charles Golumbic,
Irith Ben-Arroyo Hartman.
p. cm.
Includes bibliographical references.
ISBN-10: 0-387-24347-X ISBN-13: 978-0387-24347-4 e-ISBN 0-387-25036-0
1. Graph theory. 2. Combinatorial analysis. 3. Graph theory—Data processing.
I. Golumbic, Martin Charles. II. Hartman, Irith Ben-Arroyo.

QA166.G7167 2005
511′.5—dc22

2005042555

Copyright © 2005 by Springer Science + Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science + Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11374107

springeronline.com

Contents

Foreword ..... *vii*

Chapter 1 **Optimization Problems Related to Internet Congestion Control**
Richard Karp *1*

Chapter 2 **Problems in Data Structures and Algorithms**
Robert Tarjan *17*

Chapter 3 **Algorithmic Graph Theory and its Applications**
Martin Charles Golumbic. *41*

Chapter 4 **Decompositions and Forcing Relations in Graphs and Other Combinatorial Structures**
Ross McConnell *63*

Chapter 5 **The *Local Ratio* Technique and its Application to Scheduling and Resource Allocation Problems**
Reuven Bar-Yehuda, Keren Bendel, Ari Freund and Dror Rawitz *107*

Chapter 6 **Domination Analysis of Combinatorial Optimization Algorithms and Problems**
Gregory Gutin and Anders Yeo *145*

Chapter 7 **On Multi-Object Auctions and Matching Theory: Algorithmic Aspects**
Michal Penn and Moshe Tennenholtz *173*

Chapter 8 **Strategies for Searching Graphs**
Shmuel Gal *189*

Chapter 9 **Recent Trends in Arc Routing**
Alain Hertz *215*

Chapter 10 **Software and Hardware Testing Using Combinatorial Covering Suites**
Alan Hartman *237*

Chapter 11 **Incidences**
Janos Pach and Micha Sharir *267*

Foreword

The Haifa Workshops on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms have been held at the Caesarea Rothschild Institute (C.R.I.), University of Haifa, every year since 2001. This volume consists of survey chapters based on presentations given at the 2001 and 2002 Workshops, as well as other colloquia given at C.R.I. The Rothschild Lectures of Richard Karp (Berkeley) and Robert Tarjan (Princeton), both Turing award winners, were the highlights of the Workshops. Two chapters based on these talks are included. Other chapters were submitted by selected authors and were peer reviewed and edited. This volume, written by various experts in the field, focuses on discrete mathematics and combinatorial algorithms and their applications to real world problems in computer science and engineering. A brief summary of each chapter is given below.

Richard Karp's overview, *Optimization Problems Related to Internet Congestion Control*, presents some of the major challenges and new results related to controlling congestion in the Internet. Large data sets are broken down into smaller packets, all competing for communication resources on an imperfect channel. The theoretical issues addressed by Prof. Karp lead to a deeper understanding of the strategies for managing the transmission of packets and the retransmission of lost packets.

Robert Tarjan's lecture, *Problems in Data Structures and Algorithms*, provides an overview of some data structures and algorithms discovered by Tarjan during the course of his career. Tarjan gives a clear exposition of the algorithmic applications of basic structures like search trees and self-adjusting search trees, also known as *splay trees*. Some open problems related to these structures and to the minimum spanning tree problem are also discussed.

The third chapter by Martin Charles Golumbic, *Algorithmic Graph Theory and its Applications*, is based on a survey lecture given at Clemson University. This chapter is aimed at the reader with little basic knowledge of graph theory, and it introduces the reader to the concepts of interval graphs and other families of intersection graphs. The lecture includes demonstrations of these concepts taken from real life examples.

The chapter *Decompositions and Forcing Relations in Graphs and other Combinatorial Structures* by Ross McConnell deals with problems related to classes of intersection graphs, including interval graphs, circular-arc graphs, probe interval graphs, permutation graphs, and others. McConnell points to a general structure called *modular decomposition* which helps to obtain linear bounds for recognizing some of these graphs, and solving other problems related to these special graph classes.

In their chapter *The Local Ratio Technique and its Application to Scheduling and Resource Allocation Problems*, Bar-Yehuda, Bendel, Freund and Rawitz give a survey of the local ratio technique for approximation algorithms. An approximation algorithm efficiently finds a feasible solution to an intractable problem whose value approximates the optimum. There are numerous real life intractable problems, such as the scheduling problem, which can be approached only through heuristics or approximation algorithms. This chapter contains a comprehensive survey of approximation algorithms for such problems.

Domination Analysis of *Combinatorial Optimization Algorithms and Problems* by Gutin and Yeo provides an alternative and a complement to approximation analysis. One of the goals of domination analysis is to analyze the domination ratio of various heuristic algorithms. Given a problem P and a heuristic H, the ratio between the number of feasible solutions that are not better than a solution produced by H, and the total number of feasible solutions to P, is the domination ratio. The chapter discusses domination analyses of various heuristics for the well-known traveling salesman problem, as well as other intractable combinatorial optimization problems, such as the minimum partition problem, multiprocessor scheduling, maximum cut, k-satisfiability, and others.

Another real-life problem is the design of auctions. In their chapter *On Multi-Object Auctions and Matching Theory: Algorithmic Aspects*, Penn and Tennenholtz use b-matching techniques to construct efficient algorithms for combinatorial and constrained auction problems. The typical auction problem can be described as the problem of designing a mechanism for selling a set of objects to a set of potential buyers. In *the combinatorial auction problem* bids for bundles of goods are allowed, and the buyer may evaluate a bundle of goods for a different value than the sum of the values of each good. In *constrained auctions* some restrictions are imposed upon the set feasible solutions, such as the guarantee that a particular buyer will get at least one good from a given set. Both combinatorial and constrained auction problems are NP-complete problems, however, the authors explore special tractable instances where b-matching techniques can be used successfully.

Shmuel Gal's chapter *Strategies for Searching Graphs* is related to the problem of detecting an object such as a person, a vehicle, or a bomb hiding in a graph (on an edge or at a vertex). It is generally assumed that there is no knowledge about the probability distribution of the target's location and, in some cases, even the structure of the graph is not known. Gal uses probabilistic methods to find optimal search strategies that assure finding the target in minimum expected time.

The chapter *Recent Trends in Arc Routing* by Alain Hertz studies the problem of finding a least cost tour of a graph, with demands on the edges, using a fleet of identical vehicles. This problem and other related problems are intractable, and the chapter reports on recent exact and heuristic algorithms. The problem has applications in garbage collection, mail delivery, snow clearing, network maintenance, and many others.

Software and Hardware Testing Using Combinatorial Covering Suites by Alan Hartman is an example of the interplay between pure mathematics, computer science, and the applied problems generated by software and hardware engineers. The construction of efficient combinatorial covering suites has important applications in the testing of software and hardware systems. This chapter discusses the lower bounds on the size of covering suites, and gives a series of constructions that achieve these bounds asymptotically. These constructions involve the use of finite field theory, extremal set theory, group theory, coding theory, combinatorial recursive techniques, and other areas of computer science and mathematics.

Janos Pach and Micha Sharir's chapter, *Incidences,* relates to the following general problem in combinatorial geometry: What is the maximum number of incidences between m points and n members of a family of curves or surfaces in d-space? Results of this kind have numerous applications to geometric problems related to the distribution of distances among points, to questions in additive number theory, in analysis, and in computational geometry.

We would like to thank the authors for their enthusiastic response to the challenge of writing a chapter in this book. We also thank the referees for their comments and suggestions. Finally, this book, and many workshops, international visits, courses and projects at CRI, are the results of a generous grant from the Caesarea Edmond Benjamin de Rothschild Foundation. We are greatly indebted for their support throughout the last four years.

Martin Charles Golumbic
Irith Ben-Arroyo Hartman
Caesarea Edmond Benjamin
de Rothschild Foundation Institute for
Interdisciplinary Applications of Computer Science
University of Haifa, Israel

1

Optimization Problems Related to Internet Congestion Control

Richard Karp
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Introduction

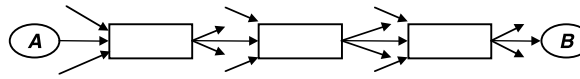
I'm going to be talking about a paper by Elias Koutsoupias, Christos Papadimitriou, Scott Shenker and myself, that was presented at the 2000 FOCS Conference [1] related to Internet-congestion control. Some people during the coffee break expressed surprise that I'm working in this area, because over the last several years, I have been concentrating more on computational biology, the area on which Ron Shamir reported so eloquently in the last lecture. I was having trouble explaining, even to myself, how it is that I've been working in these two very separate fields, until Ron Pinter just explained it to me, a few minutes ago. He pointed out to me that improving the performance of the web is crucially important for bioinformatics, because after all, people spend most of their time consulting distributed data bases. So this is my explanation, after the fact, for working in these two fields.

The Model

In order to set the stage for the problems I'm going to discuss, let's talk in slightly oversimplified terms about how information is transmitted over the Internet. We'll consider the simplest case of what's called unicast—the transmission of message or file D from one Internet host, or node, A to another node B. The data D, that host A wishes to send to host B is broken up into *packets* of equal size which are assigned consecutive serial numbers. These packets form a *flow* passing through a series of *links* and *routers* on the Internet. As the packets flow through some path of links and routers, they pass through queues. Each link has one or more queues of finite capacity in which packets are buffered as they pass through the routers. Because these buffers have a finite capacity, the queues may sometimes overflow. In that case, a choice has to be

made as to which packets shall be dropped. There are various queue disciplines. The one most commonly used, because it is the simplest, is a simple first-in-first-out (FIFO) discipline. In that case, when packets have to be dropped, the last packet to arrive will be the first to be dropped. The others will pass through the queue in first-in-first-out order.

The Internet Made Simple



- *A* wishes to send data to *B*
- *D* is broken into equal packets with consecutive serial numbers
- The packets form a flow passing through a sequence of links and routers.
- Each link has one or more queues of finite capacity.

When a packet arrives at a full queue, it is dropped.

First-in-first-out disciplines, as we will see, have certain disadvantages. Therefore, people talk about *fair queuing* where several, more complicated data structures are used in order to treat all of the data flows more fairly, and in order to transmit approximately the same number of packets from each flow. But in practice, the overhead of fair queuing is too large, although some approximations to it have been contemplated. And so, this first-in-first-out queuing is the most common queuing discipline in practical use.

Now, since not all packets reach their destination, there has to be a mechanism for the receiver to let the sender know whether packets have been received, and which packets have been received, so that the sender can retransmit dropped packets. Thus, when the receiver *B* receives the packets, it sends back an *acknowledgement* to *A*. There are various conventions about sending acknowledgements. The simplest one is when *B* simply lets *A* know the serial number of the first packet not yet received. In that case *A* will know that consecutive packets up to some point have been received, but won't know about the packets after that point which may have been received sporadically. Depending on this flow of acknowledgements back to *A*, *A* will detect that some packets have been dropped because an acknowledgement hasn't been received within a reasonable time, and will retransmit certain of these packets.

The most undesirable situation is when the various flows are transmitting too rapidly. In that case, the disaster of *congestion collapse* may occur, in which so many packets are being sent that most of them never get through—they get dropped. The acknowledgement tells the sender that the packet has been dropped. The sender sends

the dropped packet again and again, and eventually, the queues fill up with packets that are retransmissions of previous packets. These will eventually be dropped and never get to their destinations. The most important single goal of congestion control on the Internet is to avoid congestion collapse.

There are other goals as well. One goal is to give different kinds of service to different kinds of messages. For example, there are simple messages that have no particular time urgency, email messages, file transfers and the like, but then there are other kinds of flows, like streaming media etc. which have real-time requirements. I won't be getting into quality-of-service issues in this particular talk to any depth. Another goal is to allocate bandwidth fairly, so that no flow can hog the bandwidth and freeze out other flows. There is the goal of utilizing the available bandwidth. We want to avoid congestion collapse, but also it is desirable not to be too conservative in sending packets and slow down the flow unnecessarily.

The congestion control algorithm which is standard on the Internet is one that the various flows are intended to follow voluntarily. Each flow under this congestion control algorithm has a number of parameters. The most important one is the window size W—the maximum number of packets that can be in process; more precisely, W is the maximum number of packets that the sender has sent but for which an acknowledgement has not yet been received. The second parameter of importance is the roundtrip time (RTT). This parameter is a conservative upper estimate on the time it should take for a packet to reach its destination and for the acknowledgement to come back. The significance of this parameter is that if the acknowledgement is not received within RTT time units after transmission, then the sender will assume that the packet was dropped. Consequently, it will engage in retransmission of that particular packet and of all the subsequent packets that were sent up to that point, since packet drops often occur in bursts.

In the ideal case, things flow smoothly, the window size is not excessive and not too small, no packet is dropped, and A receives an acknowledgement and sends a packet every RTT/W time steps. But in a bad case, the packet "times out", and then all packets sent in the last interval of time RTT must be retransmitted. The crucial question is, therefore, how to modify, how to adjust this window. The window size should continually increase as long as drops are not experienced, but when drops are experienced, in order to avoid repetition of those drops, the sender should decrease its window size.

The Jacobson algorithm, given below, is the standard algorithm for adjusting the window size. All Internet service providers are supposed to adhere to it.

Jacobson's Algorithm for adjusting W

start-up:

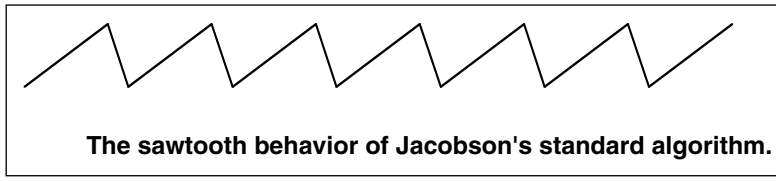
$W \leftarrow 1$

when acknowledgment received

$W \leftarrow W + 1$

```
        when timeout occurs
            W ← ⌊W/2⌋
            go to main
main:
        if W acknowledgements received before timeout occurs then
            W ← W + 1
        else
            W ← ⌊W/2⌋
```

Jacobson's algorithm gives a rather jagged behavior over time. The window size W is linearly increased, but from time to time it is punctuated by a sudden decrease by a factor of two. This algorithm is also called the *additive increase/multiplicative decrease (AIMD) scheme*. There are a number of variations and refinements to this algorithm. The first variation is called *selective acknowledgement*. The acknowledgement is made more informative so that it indicates not only the serial number of the first packet not yet received, but also some information about the additional packets that have been received out of order.



The sawtooth behavior of Jacobson's standard algorithm.

The second variation is "*random early drop*." The idea is that instead of dropping packets only when catastrophe threatens and the buffers start getting full, the packets get dropped randomly as the buffers approach saturation, thus giving an early warning that the situation of packet dropping is approaching. Another variation is *explicit congestion notification*, where, instead of dropping packets prematurely at random, warnings are issued in advance. The packets go through, but in the acknowledgement there is a field that indicates "you were close to being dropped; maybe you'd better slow down your rate." There are other schemes that try to send at the same long-term average rate as Jacobson's algorithm, but try to smooth out the flow so that you don't get those jagged changes, the abrupt decreases by a factor of two.

The basic philosophy behind all the schemes that I've described so far is voluntary compliance. In the early days, the Internet was a friendly club, and so you could just ask people to make sure that their flows adhere to this standard additive increase/multiplicative decrease (AIMD) scheme. Now, it is really social pressure that holds things together. Most people use congestion control algorithms that they didn't implement themselves but are implemented by their service provider and if their service provider doesn't adhere to the AIMD protocol, then the provider gets a bad reputation. So they tend to adhere to this protocol, although a recent survey of the actual algorithms provided by the various Internet service providers indicates a considerable amount of

deviation from the standard, some of this due to inadvertent program bugs. Some of this may be more nefarious—I don't know.

In the long run, it seems that the best way to ensure good congestion control is not to depend on some voluntary behavior, but to induce the individual senders to moderate their flows out of self-interest. If no reward for adhering, or punishment for violation existed, then any sender who is motivated by self-interest could reason as follows: what I do has a tiny effect on packet drops because I am just one of many who are sharing these links, so I should just send as fast as I want. But if each individual party follows this theme of optimizing for itself, you get the "tragedy of the commons", and the total effect is a catastrophe. Therefore, various mechanisms have been suggested such as: monitoring individual flow rates, or giving flows different priority levels based on pricing.

The work that we undertook is intended to provide a foundation for studying how senders should behave, or could be induced to behave, if their goal is self-interest and they cannot be relied on to follow a prescribed protocol. There are a couple of ways to study this. We have work in progress which considers the situation as an n-person non-cooperative game. In the simplest case, you have n flows competing for a link. As long as some of their flow rates are below a certain threshold, everything will get through. However, as soon as the sum of their flow rates crosses the threshold, some of them will start experiencing packet drops. One can study the Nash equilibrium of this game and try to figure out different kinds of feedback and different kinds of packet drop policies which might influence the players to behave in a responsible way.

The Rate Selection Problem

In the work that I am describing today, I am not going to go into this game theoretic approach, which is in its preliminary stages. I would like to talk about a slightly different situation. The most basic question one could perhaps ask is the following: suppose you had a single flow which over time is transmitting packets, and the flow observes that if it sends at a particular rate it starts experiencing packet drops; if it sends at another rate everything gets through. It gets this feedback in the form of acknowledgements, and if it's just trying to optimize for itself, and is getting some partial information about its environment and how much flow it can get away with, how should it behave?

The formal problem that we will be discussing today is called the *Rate Selection Problem*. The problem is: how does a single, self-interested host A, observing the limits on what it can send over successive periods of time, choose to moderate its flow. In the formal model, time will be divided into intervals of fixed length. You can think of the length of the interval as perhaps the roundtrip time. For each time interval t there is a parameter u_t, defined as the maximum number of packets that A can send B without experiencing packet drops. The parameter u_t is a function of all the other flows in the system, of the queue disciplines that are used, the topology of the Internet, and other factors. Host A has no direct information about u_t. In each time interval t, the parameter x_t denotes the number of packets sent by the sender A. If $x_t \leq u_t$, then

all the packets will be received, none of them will time out and everything goes well. If $x_t > u_t$, then at least one packet will be dropped, and the sender will suffer some penalty that we will have to model. We emphasize that the sender does not have direct information about the successive thresholds. The sender only gets partial feedback, i.e. whether $x_t \leq u_t$ or not, because all that the sender can observe about the channel is whether or not drops occurred.

In order to formulate an optimization problem we need to set up a cost function $c(x,u)$. The function represents the cost of transmitting x packets in a time period with threshold u. In our models, the cost reflects two major components: *opportunity cost* due to sending of less than the available bandwidth, i.e. when $x_t < u_t$, and *retransmission delay and overhead* due to dropped packets when $x_t > u_t$.

We will consider here two classes of cost functions.

The ***severe cost function*** is defined as follows:

$$c(x_t, u_t) = \begin{cases} u_t - x_t & \text{if } x_t \leq u_t \\ u_t & \text{otherwise} \end{cases}$$

The intuition behind this definition is the following: When $x_t \leq u_t$, the user pays the difference between the amount it could have sent and the actual amount sent. When $x_t > u_t$, we'll assume the sender has to resend all the packets that it transmitted in that period. In that case it has no payoff for that period and its cost is u_t, because if it had known the threshold, it could have got u_t packets through, but in fact, it gets zero.

The ***gentle cost function*** will be defined as:

$$c(x_t, u_t) = \begin{cases} u_t - x_t & \text{if } x_t \leq u_t \\ \alpha(x_t - u_t) & \text{otherwise} \end{cases}$$

where α is a fixed proportionality factor. Under this function, the sender is punished less for slightly exceeding the threshold. There are various interpretations of this. In certain situations it is not strictly necessary for all the packets to get through. Only the quality of information received will deteriorate. Therefore, if we assume that the packets are not retransmitted, then the penalty simply relates to the overhead of handling the extra packets plus the degradation of the quality at the receiver. There are other scenarios when certain erasure codes are used, where it is not a catastrophe not to receive certain packets, but you still pay an overhead for sending too many packets. Other cost functions could be formulated but we will consider only the above two classes of cost functions.

The ***optimization problem*** then is the following: Choose over successive periods the amounts x_t of packets to send, so as to minimize the total cost incurred over all periods. The amount x_{t+1} is chosen knowing the sequence $x_1, x_2, \ldots, x_t$ and whether $x_i \leq u_i$ or not, for each $i = 1, 2, \ldots, t$.

The Static Case

We begin by investigating what we call the ***static case***, where the conditions are unchanging. In the static case we assume that the threshold is fixed and is a positive integer less than or equal to a known upper bound n, that is, $u_t = u$ for all t, where $u \in \{1, 2, \ldots, n\}$. At step t, A sends x_t packets and learns whether $x_t \leq u_t$. The problem can be viewed as a Twenty Questions game in which the goal is to determine the threshold u at minimum cost by queries of the form, "Is $x_t > u$?" We remark that the static case is not very realistic. We thought that we would dispose of it in a few days, and move on to the more interesting dynamic case. However, it turned out that there was a lot of mathematical content even to the static case, and the problem is rather nice. We give below an outline of some of the results.

At step t of the algorithm, the sender sends an amount x_t, pays a penalty $c(x_t, u_t)$ according to whether x_t is above or below the threshold, and gets feedback telling it whether $x_t \leq u_t$ or not. At a general step, there is an *interval of pinning* containing the threshold. The initial interval of pinning is the interval from *1* to n. We can think of an algorithm for determining the threshold as a function from intervals of pinning to integers. In other words, for every interval of pinning $[i, j]$, the algorithm chooses a flow k, $(i \leq k \leq j)$ for the next interval. The feedback to this flow will tell the sender whether k was above the threshold or not. In the first case, there will be packet drops and the next interval of pinning will be the interval $[i, k-1]$. In the second case, the sender will succeed in sending the flow through, there will be no packet drops, and the interval of pinning at the next time interval will be the interval $[k, j]$. We can thus think of the execution of the algorithm as a decision tree related to a twenty questions game attempting to identify the actual threshold. If the algorithm were a simple binary search, where one always picks the middle of the interval of pinning, then the tree of Figure 1 would represent the possible runs of the algorithm. Each leaf of the tree corresponds to a possible value of the threshold. Let $A(u)$ denote the cost of the algorithm A, when the

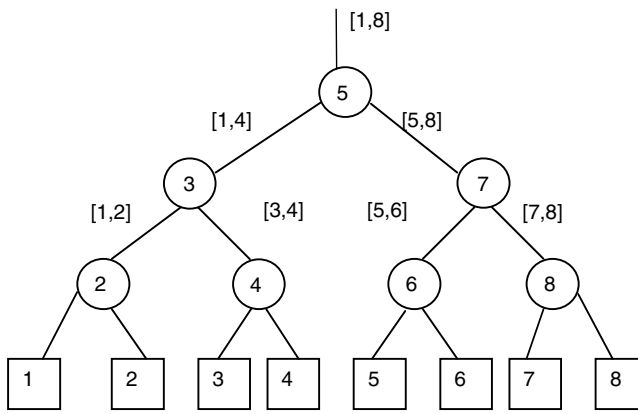


Figure 1.

threshold is u. We could be interested in the *expected cost* which is the average cost over all possible values of the threshold, i.e. $1/n \sum_{u=1}^{n} A(u)$. We could also be interested in the *worst-case costs*, i.e. $\max_{1 \le u \le n} A(u)$. For the different cost functions defined above, ("gentle" and "severe") we will be interested in algorithms that are optimal either with respect to the expected cost or with respect to the worst-case cost.

It turns out that for an arbitrary cost function $c(x, u)$, there is a rather simple dynamic programming algorithm with running time $O(n^3)$, which minimizes expected cost. In some cases, an extension of dynamic programming allows one to compute policies that are optimal in the worst-case sense. So the problem is not so much computing the optimal policy for a particular value of the upper limit and of the threshold, but rather of giving a nice characterization of the policy. It turns out that for the gentle cost function family, for large n, there is a very simple characterization of the optimal policies. And this rule is essentially optimal in an asymptotic sense with respect to both the expected cost and the worst-case cost.

The basic question is: Given an interval of pinning $[i, j]$, where should you put your next question, your next transmission range. Clearly, the bigger α is, the higher the penalty for sending too much, and the more cautious one should be. For large α we should put our trial value close to the beginning of the interval of pinning in order to avoid sending too much. It turns out that the optimal thing to do asymptotically is always to divide the interval of pinning into two parts in the proportions $1{:}\sqrt{\alpha}$. The expected cost of this policy is $\sqrt{\alpha}\, n/2 + O(\log n)$ and the worst-case cost is $\sqrt{\alpha}\, n + O(\log n)$. Outlined proofs of these results can be found in [1].

These results can be compared to binary search, which has expected cost $(1 + \alpha)n/2$. Binary search does not do as well, except in the special case where $\alpha = 1$, in which case the policy is just to cut the interval in the middle.

So that's the complete story, more or less, of the gentle-cost function in the static case. For the severe-cost function, things turn out to be more challenging.

Consider the binary search tree as in Figure 2, and assume that $n = 8$ and the threshold is $u = 6$. We would start by trying to send 5 units. We would get everything through but we would pay an opportunity cost of 1. That would take us to the right child of the root. Now we would try to send 7 units. Seven is above the threshold 6, so we would overshoot and lose 6, and our total cost thus far would be $1 + 6$. Then we would try 6, which is the precise threshold. The information that we succeeded would be enough to tell us that the threshold was exactly 6, and thereafter we would incur no further costs. So we see that in this particular case the cost is 7. Figure 2 below demonstrates the costs for each threshold u (denoted by the leaves of the tree). The total cost in this case is 48, the expected cost is 48/8, the worst-case cost is 10. It turns out that for binary search both the expected cost and the worst-case cost are $O(n \log n)$.

The question is, then, can we do much better than $O(n \log n)$? It turns out that we can. Here is an algorithm that achieves $O(n \log \log n)$. The idea of this algorithm is as

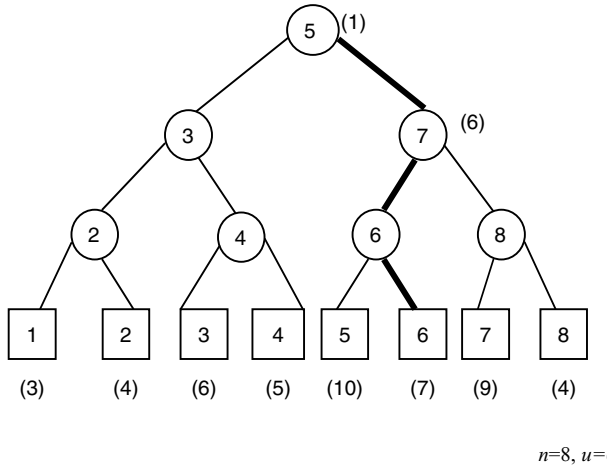


Figure 2.

follows: The algorithm runs in successive phases. Each phase will have a target—to reduce the interval of pinning to a certain size. These sizes will be, respectively, $n/2$ after the first phase, $n/2^2$ after the second phase, $n/2^4$ after the third phase, $n/2^8$ after the 4^{th} phase, and $n/2^{2^{k-1}}$ after the k-th phase. It's immediate then that the number of phases will be $1 + \log\log n$, or $O(\log\log n)$.We remark that we are dealing with the severe-cost function where there is a severe penalty for overshooting, for sending too much. Therefore, the phases will be designed in such a way that we overshoot at most once per phase.

We shall demonstrate the algorithm by a numerical example. Assume $n = 256$ and the threshold is $u = 164$. In each of the first two phases, it is just like binary search. We try to send 128 units. We succeed because $128 \leq 164$. Now we know that the interval of pinning is [128, 256]. We try the midpoint of the interval, 192. We overshoot. Now the interval of pinning is of length 64. At the next step we are trying to reduce the interval of pinning down to 16, which is 256 over 2^4. We want to be sure of overshooting only once, so we creep up from 128 by increments of 16. We try 144; we succeed. We try 160; we succeed. We try 176; we fail. Now we know that the interval of pinning is [160, 175]. It contains 16 integers. At the next stage we try to get an interval of pinning of size 1. We do so by creeping up one at a time, 161, 162, etc. until we reach the correct threshold $u = 164$. A simple analysis shows that the cost of each phase is $O(n)$, and since the number of phases is $O(\log\log n)$, the cost of the algorithm is $O(n \log\log n)$.

A Lower Bound

The question then is, is it possible to improve the bound $O(n \log\log n)$? The answer is negative as is seen in the next theorem.

Theorem 1 $\min_{A} \max_{1 \le u \le n} A(u) = \Theta(n \log\log n)$.

Theorem 1 claims that the best complexity of an algorithm, with a given a priori bound on the threshold $u \le O(n)$, is $\Theta(n \log\log n)$. This is achievable by the algorithm described above.

There is also another result that deals with the case where no upper bound is given on the threshold. In this case, as well, a bound of $\Theta(u \log\log u)$ is achieved for every threshold u.

We shall demonstrate the idea behind the proof of the lower bound in Theorem 1. Any run of an algorithm corresponds to some path from the root to a leaf in the binary decision tree. The path contains right and left turns. A right turn means that the amount we send is less than or equal to the threshold; a left turn means that we overshoot, and the amount that we send is greater than the threshold. The left turns are very undesirable because we lose an amount equal to the threshold whenever we take a left turn. However, we also accumulate costs associated with the right turns, because we are not sending as much as we could have. We therefore have a trade-off between the number of left turns, and the cost of right turns. For threshold u denote the number of left turns in the path from root to leaf u by $\mathit{leftheight}(u)$. Let $\mathit{rightcost}(u)$ denote the sum of the costs accumulated in the right turns. Thus, the cost of an algorithm is given by

$$A(u) = u \cdot \mathit{leftheight}(u) + \mathit{rightcost}(u)$$

For example, for the path given in Figure 3 we have $\mathit{leftheight}(15) = 2$ and $\mathit{rightcost}(15) = (15 - 7) + (15 - 13) + (15 - 14) = 11$.

We define two more parameters related to the binary tree T. Let $\mathit{leftheight}(T) = \max_{u} \mathit{leftheight}(u)$, and $\mathit{rightcost}(T) = \sum_{u} \mathit{rightcost}(u)$.

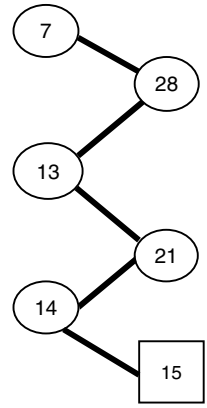
The following key lemma states that there is an inherent antagonism between minimizing the left height and the goal of minimizing the right cost.

Lemma 1 *There exists a constant $a > 0$ such that every n-leaf binary tree T with $\mathit{leftheight}(T) \le \log\log n$ has $\mathit{rightcost}(T) \ge an^2 \log\log n$.*

The proof of Theorem 1 now follows easily from Lemma 1. For details see [1].

The Dynamic Case

So far we have discussed the static problem, which is not entirely realistic. The static problem means that the sender is operating under constant conditions, but we don't expect that to be the case. We expect some fluctuation in the rate available to the sender from period to period.



Leftheight(15)=2

Rightcost(15)=(15-7)+(15-13)+(15-14)

Figure 3.

In the dynamic case, you can think of an adversary who is changing the threshold in such a way as to fool the sender. The problem has different forms depending on the restrictions we assume on the adversary. If the adversary can just do anything it would like in any period, then clearly the sender doesn't have a clue what to do. So we may have various assumptions on the adversary. We can assume that the threshold u_t, chosen by the adversary, is simply an integer satisfying $u_t \in [a, b]$ where a and b are any two integers. Or we can assume that the variation of the threshold is more restricted. One such assumption that we investigated is that the adversary can drop the threshold as rapidly as it likes but can only increase the threshold from one period to the next by at most a factor, $\theta > 1$, i.e. $u_{t+1} \in [0, \theta u_t]$. Another possible assumption is that the threshold is bounded below by a positive constant β and the adversary is additively constrained so that it can only increase the threshold by some fixed amount, α, at most in any period, i.e. $u_{t+1} \in [\beta,\ u_t + \alpha]$.

As in the static case, the game is played in rounds, where in each round the algorithm sends x_t packets. Unlike the static case, here we assume that the adversary chooses a sequence $\{u_t\}$ of thresholds by knowing the algorithm for choosing the sequence $\{x_t\}$ of probes. Up to this point, we have considered the cost or the loss that the sender has. Now we are going to consider the *gain* that the player achieves. The gain is defined as $g(x_t, u_t) = u_t - c(x_t, u_t)$, where $c(x_t, u_t)$ is the severe cost function. It is essentially the number of packets that the player gets through. The player receives feedback $f(x_t, u_t)$ which is a single bit stating whether or not the amount sent is less than or equal to the threshold for the current period.

Why are we suddenly switching from lose to gain? This is, after all, an online problem. The sender is making adaptive choices from period to period, making each

choice on the basis of partial information from the previous period. The traditional approach for analyzing online problems is of *competitive analysis* [2], in which the performance of an on-line algorithm for choosing $\{x_t\}$ is compared with the best among some family of off-line algorithms for choosing $\{x_t\}$. An off-line algorithm knows the entire sequence of thresholds $\{u_t\}$ beforehand. An unrestricted off-line algorithm could simply choose $x_t = u_t$ for all t, incurring a total cost of zero. The ratio between the on-line algorithm's cost and that of the off-line algorithm would then be infinite, and could not be used as a basis for choosing among on-line algorithms. For this reason it is more fruitful to study the gain rather than the loss.

The algorithm's gain (ALG) is defined as the sum of the gains over the successive periods, and the adversary's gain (OPT) is the sum of the thresholds because the omniscient adversary would send the threshold amount at every step.

We adopt the usual definition of a randomized algorithm. We say that a randomized algorithm achieves *competitive ratio r* if for every sequence of thresholds.

$r \cdot ALG \geq OPT + const$, where *const* depends only on the initial conditions.

This means that, for every oblivious adversary, its payoff is a fraction 1/r of the amount that the adversary could have gotten. By an *oblivious* adversary we mean an adversary which knows the general policy of the algorithm, but not the specific random bits that the algorithm may generate from step to step. It has to choose the successive thresholds in advance, just knowing the text of the algorithm, but not the random bits generated by the algorithm. If the algorithm is deterministic, then the distinction between oblivious adversaries and general adversaries disappears.

We have a sequence of theorems about the optimal competitive ratio. We will mention them briefly without proofs. The proofs are actually, as is often the case with competitive algorithms, trivial to write down once you have guessed the answer and come up with the right potential function. For those who work with competitive algorithms this is quite standard.

Adversary Restricted to a Fixed Interval

The first case we consider is when the adversary can be quite wild. It can choose any threshold u_t from a fixed interval $[a, b]$. The deterministic case is trivial: An optimal on-line algorithm would never select a rate $x_t > a$ because of the adversary's threat to select $u_t = a$. But if the algorithm transmits at the minimum rate $x_t = a$, the adversary will select the maximum possible bandwidth $u_t = b$, yielding a competitive ratio of b/a. If randomization is allowed then the competitive ratio improves, as is seen in the following theorem:

Theorem 2 *The optimal randomized competitive ratio against an adversary that is constrained to select* $u_t \in [a, b]$ *is* $1 + \ln(b/a)$.

The analysis of this case is proved by just considering it as a two-person game between the algorithm and the adversary and giving optimal mixed strategies for the two players. The details are given in [1].

Adversary Restricted by a Multiplicative Factor

It is more reasonable to suppose that the adversary is multiplicatively constrained. In particular, we assume that the adversary can select any threshold $u_{t+1} \in [0\ ,\ \theta\ u_t]$ for some constant $\theta \geq 1$. The adversary can only increase the threshold by, at most, some factor θ, from one period to the next. You might imagine that we would also place a limit on how much the adversary could reduce the threshold but it turns out we can achieve just as good a competitive ratio without this restriction. It would be nice if it turned out that an optimal competitive algorithm was additive-increase/multiplicative-decrease. That this would give a kind of theoretical justification for the Jacobson algorithm, the standard algorithm that is actually used. But we haven't been quite so lucky. It turns out that if you are playing against the multiplicatively constrained adversary, then there's a nearly optimal competitive algorithm which is of the form multiplicative-increase/multiplicative-decrease. The result is stated below:

Theorem 3 *There is a deterministic online algorithm with competitive ratio* $(\sqrt{\theta} + \sqrt{\theta - 1})^2$ *against an adversary who is constrained to select any threshold* u_{t+1} *in the range* $[0, \theta\, u_t]$ *for some constant* $\theta \geq 1$. *On the other hand, no deterministic online algorithm can achieve a competitive ratio better than* θ.

In the proof, the following ***multiplicative-increase/multiplicative-decrease*** algorithm is considered: If you undershoot, i.e. if $x_t \leq u_t$

$$\textbf{\textit{then}}\ x_{t+1} = \theta\, x_t$$

$$\textbf{\textit{else}}\ x_{t+1} = \lambda\, x_t, \textit{ where } \lambda = \frac{\sqrt{\theta}}{\sqrt{\theta} + \sqrt{\theta - 1}}$$

It is argued in [1] that the following two invariants are maintained:

- $u_t \leq \dfrac{\theta}{\lambda} x_t$, and
- $rgain_t \geq opt_t + \Phi(x_{t+1}) - \Phi(x_1)$, where $\Phi(x) = \dfrac{1}{1 - \lambda} x$ is an appropriate *potential function.*

Once the right policy, the right bounds, and the right potential function are guessed, then the theorem follows from the second invariant using induction. I should say that most of this work on the competitive side was done by Elias Koutsoupias.

Adversary Restricted by an Additive Term

We consider the case where the adversary is bounded below by a positive constant and constrained by an additive term, i,e, $u_{t+1} \in [\beta, u_t + \alpha]$. For a multiplicatively constrained adversary you get a multiplicative-increase/ multiplicative-decrease algorithm. You might guess that for an additively constrained adversary you get additive-increase/additive–decrease algorithm. That's in fact what happens:

Theorem 4 *The optimal deterministic competitive ratio against an adversary constrained to select threshold u_{t+1} in the interval $[\beta, u_t + \alpha]$ is at most $4 + \alpha/\beta$. On the other hand, no deterministic online algorithm has competitive ratio better than $1 + \alpha/\beta$.*

The algorithm is a simple additive-increase/additive–decrease algorithm and again the proof involves certain inductive claims that, in turn, involve a potential function that has to be chosen in exactly the right way. For more details, consult the paper [1].

There is a very nice development that came out this somewhat unexpectedly and may be of considerable importance, not only for this problem, but also for others. I went down to Hewlett-Packard and gave a talk very much like this one. Marcello Weinberger at Hewlett-Packard asked, "Why don't you formulate the problem in a different way, taking a cue from work that has been done in information theory and economics on various kinds of prediction problems? Why don't you allow the adversary to be very free to choose the successive thresholds any way it likes, from period to period, as long as the thresholds remain in the interval $[a, b]$? But don't expect your algorithm to do well compared to arbitrary algorithms. Compare it to a reasonable class of algorithms." For example, let's consider those algorithms which always send at the same value, but do have the benefit of hindsight. So the setting was that we will allow the adversary to make these wild changes, anything in the interval $[a, b]$ at every step, but the algorithm only has to compete with algorithms that send the same amount in every period.

This sounded like a good idea. In fact, this idea has been used in a number of interesting studies. For example, there is some work from the 70's about the following problem: Suppose your adversary is choosing a sequence of heads and tails and you are trying to guess the next coin toss. Of course, it's hopeless because if the adversary knows your policy, it can just do the opposite. Yet, suppose you are only trying to compete against algorithms which know the whole sequence of heads and tails chosen by the adversary but either have to choose heads all the time or have to choose tails all the time. Then it turns out you can do very well even though the adversary is free to guess what you are going to do and do the opposite; nevertheless you can do very well against those two extremes, always guessing heads and always guessing tails.

There is another development in economics, some beautiful work by Tom Cover, about an idealized market where there is no friction, no transaction costs. He shows

that there is a way of changing your portfolio from step to step, which of course cannot do well against an optimal adaptive portfolio but can do well against the best possible fixed market basket of stocks even if that market basket is chosen knowing the future course of the market.

There are these precedents for comparing your algorithm against a restricted family of algorithms, even with a very wild adversary. I carried this work back to ICSI where I work and showed it Antonio Piccolboni and Christian Schindelhauer. They got interested in it. Of course, the hallmark of our particular problem is that unlike these other examples of coin tossing and the economic market basket, in our case, we don't really find out what the adversary is playing. We only get limited feedback about the adversary, namely, whether the adversary's threshold was above or below the amount we sent. Piccolboni and Schindelhauer undertook to extend some previous results in the field by considering the situation of limited feedback. They considered a very general problem, where in every step the algorithm has a set of moves, and the adversary has a set of moves. There is a loss matrix indicating how much we lose if we play i and the adversary plays j. There is a feedback matrix which indicates how much we find out about what the adversary actually played, if we play i and if the adversary plays j.

Clearly, our original problem can be cast in this framework. The adversary chooses a threshold. The algorithm chooses a rate. The loss is according to whether we overshoot or undershoot and the feedback is either 0 or 1, according to whether we overshoot or undershoot. This is the difference from the classical results of the 1970's. We don't really find out what the adversary actually played. We only find out partial information about what the adversary played.

The natural measure of performance in this setting is worst-case regret. What it is saying is that we are going to compare, in the worst-case over all choices of the successive thresholds by the adversary, our expected loss against the minimum loss of an omniscient player who, however, always has to play the same value at every step. The beautiful result is that, subject to a certain technical condition which is usually satisfied, there will be a randomized algorithm even in the case of limited feedback which can keep up with this class of algorithms, algorithms that play a constant value, make the same play at every step. This is very illuminating for our problem, but we think that it also belongs in the general literature of results about prediction problems and should have further applications to statistical and economic games. This is a nice side effect to what was originally a very specialized problem.

Acknowledgment

The author wishes to express his admiration and sincere gratitude to Dr. Irith Hartman for her excellent work in transcribing and editing the lecture on which this paper is based.

References

[1] R. Karp, E. Koutsoupias, C. Papadimitriou, and S. Shenker. Combinatorial optimization in congestion control. In *Proceedings of the 41th Annual Symposium on Foundations of Computer Science*, pp. 66–74, Redondo Beach, CA, 12–14 November (2000).

[2] A. Borodin and R. El-Yaniv. Online Computation and Competitive Analysis. Cambridge University Press (1998).

2

Problems in Data Structures and Algorithms

Robert E. Tarjan
Princeton University and Hewlett Packard

1. Introduction

I would like to talk about various problems I have worked on over the course of my career. In this lecture I'll review simple problems with interesting applications, and problems that have rich, sometimes surprising, structure.

Let me start by saying a few words about how I view the process of research, discovery and development. (See Figure 1.)

My view is based on my experience with data structures and algorithms in computer science, but I think it applies more generally. There is an interesting interplay between theory and practice. The way I like to work is to start out with some application from the real world. The real world, of course, is very messy and the application gets modeled or abstracted away into some problem or some setting that someone with a theoretical background can actually deal with. Given the abstraction, I then try to develop a solution which is usually, in the case of computer science, an algorithm, a computational method to perform some task. We may be able to prove things about the algorithm, its running time, its efficiency, and so on. And then, if it's at all useful, we want to apply the algorithm back to the application and see if it actually solves the real problem. There is an interplay in the experimental domain between the algorithm developed, based on the abstraction, and the application; perhaps we discover that the abstraction does not capture the right parts of the problem; we have solved an interesting mathematical problem but it doesn't solve the real-world application. Then we need to go back and change the abstraction and solve the new abstract problem and then try to apply that in practice. In this entire process we are developing a body of new theory and practice which can then be used in other settings.

A very interesting and important aspect of computation is that often the key to performing computations efficiently is to understand the problem, to represent the

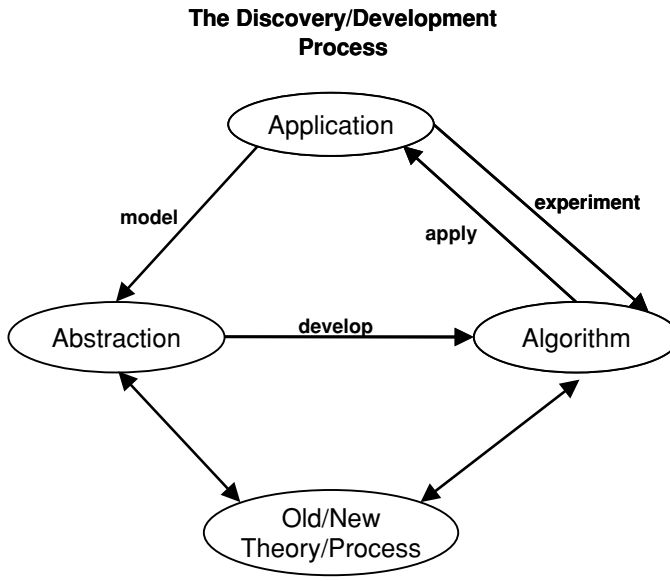


Figure 1.

problem data appropriately, and to look at the operations that need to be performed on the data. In this way many algorithmic problems turn into data manipulation problems, and the key issue is to develop the right kind of data structure to solve the problem. I would like to talk about several such problems. The real question is to devise a data structure, or to analyze a data structure which is a concrete representation of some kind of algorithmic process.

2. Optimum Stack Generation Problem

Let's take a look at the following simple problem. I've chosen this problem because it's an abstraction which is, on the one hand, very easy to state, but on the other hand, captures a number of ideas. We are given a finite alphabet Σ, and a stack S. We would like to generate strings of letters over the alphabet using the stack. There are three stack operations we can perform.

push (A)—push the letter **A** from the alphabet onto the stack,
emit—output the top letter from the stack,
pop—pop the top letter from the stack.

We can perform any sequence of these operations subject to the following well-formedness constraints: we begin with an empty stack, we perform an arbitrary series of ***push***, ***emit*** and ***pop*** operations, we never perform ***pop*** from an empty stack, and we

end up with an empty stack. These operations generate some sequence of letters over the alphabet.

Problem 2.1 *Given some string σ over the alphabet, find a minimum length sequence of stack operations to generate σ.*

We would like to find a fast algorithm to find the minimum length sequence of stack operations for generating any particular string.

For example, consider the string **A B C A C B A.** We could generate it by performing: ***push*** **(A),** ***emit*** **A,** ***pop*** **A,** ***push*** **(B),** ***emit*** **B,** ***pop*** **B,** ***push*** **(C),** ***emit*** **C,** ***pop*** **C** etc., but since we have repeated letters in the string we can use the same item on the stack to generate repeats. A shorter sequence of operations is: ***push*** **(A),** ***emit*** **A,** ***push*** **(B),** ***emit*** **B,** ***push*** **(C),** **e*mit*** **C,** ***push*** **(A),** ***emit*** **A,** ***pop*** **A**; now we can ***emit*** **C** (we don't have to put a new **C** on the stack), ***pop*** **C,** ***emit*** **B,** ***pop*** **B,** ***emit*** **A**. We got the '**CBA**' string without having to do additional push-pops. This problem is a simplification of the programming problem which appeared in "The International Conference on Functional Programming" in 2001 [46] which calls for optimum parsing of HTML-like expressions.

What can we say about this problem? There is an obvious $O(n^3)$ dynamic programming algorithm[1]. This is really a special case of optimum context-free language parsing, in which there is a cost associated with each rule, and the goal is to find a minimum-cost parse. For an alphabet of size three there is an $O(n)$ algorithm (Y. Zhou, private communication, 2002). For an alphabet of size four, there is an $O(n^2)$ algorithm. That is all I know about this problem. I suspect this problem can be solved by matrix multiplication, which would give a time complexity of $O(n^\alpha)$, where α is the best exponent for matrix multiplication, currently 2.376 [8]. I have no idea whether the problem can be solved in $O(n^2)$ or in $O(n \log n)$ time. Solving this problem, or getting a better upper bound, or a better lower bound, would reveal more information about context-free parsing than what we currently know. I think this kind of question actually arises in practice. There are also string questions in biology that are related to this problem.

3. Path Compression

Let me turn to an old, seemingly simple problem with a surprising solution. The answer to this problem has already come up several times in some of the talks in

[1] *Sketch of the algorithm*: Let $S[1 \ldots n]$ denote the sequence of characters. Note that there must be exactly n emits and that the number of pushes must equal the number of pops. Thus we may assume that the cost is simply the number of pushes. The dynamic programming algorithm is based on the observation that if the same stack item is used to produce, say, $S[i_1]$ and $S[i_2]$, where $i_2 > i_1$ and $S[i_1] = S[i_2]$, then the state of the stack at the time of emit $S[i_1]$ must be restored for emit $S[i_2]$. Thus the cost $C[i, j]$ of producing the subsequence $S[i, j]$ is the minimum of $C[i, j-1] + 1$ and $\min\{C[i, t] + C[t+1, j-1] : S[t] = S[j],\ i \le t < j\}$.

the conference "Second Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms." The goal is to maintain a collection of n elements that are partitioned into sets, i.e., the sets are always disjoint and each element is in a unique set. Initially each element is in a singleton set. Each set is named by some arbitrary element in it. We would like to perform the following two operations:

find(x)—for a given arbitrary element x, we want to return the name of the set containing it.

unite(x,y)—combine the two sets named by x and y. The new set gets the name of one of the old sets.

Let's assume that the number of elements is n. Initially, each element is in a singleton set, and after $n - 1$ ***unite*** operations all the elements are combined into a single set.

Problem 3.1 *Find a data structure that minimizes the worst-case total cost of m* **find** *operations intermingled with n − 1 unite operations.*

For simplicity in stating time bounds, I assume that $m \geq n$, although this assumption is not very important. This problem originally arose in the processing of COMMON and EQUIVALENCE statements in the ancient programming language FORTRAN. A solution is also needed to implement Kruskal's [31] minimum spanning tree algorithm. (See Section 7.)

There is a beautiful and very simple algorithm for solving Problem 3.1, developed in the '60s. I'm sure that many of you are familiar with it. We use a forest data structure, with essentially the simplest possible representation of each tree (see Figure 2). We use rooted trees, in which each node has one pointer, to its parent. Each set is represented by a tree, whose nodes represent the elements, one element per node. The root element is the set name. To answer a ***find(x)*** operation, we start at the given node x and follow

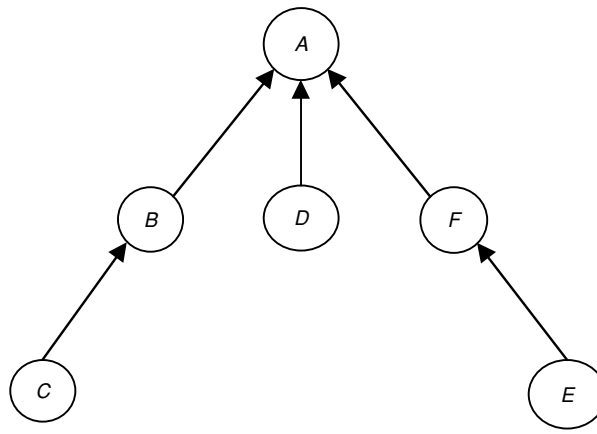


Figure 2.

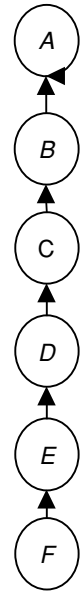


Figure 3.

the pointers to the root node, which names the set. The time of the ***find*** operation is proportional to the length of the path. The tree structure is important here because it affects the length of the ***find*** path. To perform a ***unite(x,y)*** operation, we access the two corresponding tree roots ***x*** and ***y***, and make one of the roots point to the other root. The ***unite*** operation takes constant time.

The question is, how long can ***find*** paths be? Well, if this is all there is to it, we can get bad examples. In particular, we can construct the example in Figure 3: a tree which is just a long path. If we do lots of ***finds***, each of linear cost, then the total cost is proportional to the number of ***finds*** times the number of elements, $O(m \cdot n)$, which is not a happy situation.

As we know, there are a couple of heuristics we can add to this method to substantially improve the running time. We use the fact that the structure of each tree is completely arbitrary. The best structure for the ***find*****s** would be if each tree has all its nodes just one step away from the root. Then ***find*** operations would all be at constant cost. But as we do the ***unite*** operations, depths of nodes grow. If we perform the ***unites*** intelligently, however, we can ensure that depths do not become too big. I shall give two methods for doing this.

Unite **by size** (Galler and Fischer [16]): This method combines two trees into one by making the root of the smaller tree point to the root of the larger tree (breaking a tie arbitrarily). The method is described in the pseudo-code below. We maintain with each root ***x*** the tree size, ***size***(***x***) (the number of nodes in the tree).

unite(x,y): If ***size (x)*** $\geq$ ***size (y)*** make ***x*** the parent of ***y*** and set

size (x) $\leftarrow$ ***size (x)*** $+$ ***size (y)***

Otherwise make ***y*** the parent of ***x*** and set

size (y) $\leftarrow$ ***size (x)*** $+$ ***size (y)***

Unite* by rank** (Tarjan and van Leewen [41]): In this method each root contains a ***rank, which is an estimate of the depth of the tree. To combine two trees with roots of different rank, we attach the tree whose root has smaller rank to the other tree, without changing any ranks. To combine two trees with roots of the same rank, we attach either tree to the other, and increase the rank of the new root by one. The pseudo code is below. We maintain with each root ***x*** its rank, ***rank(x)***. Initially, the rank of each node is zero.

unite(x, y): if ***rank (x)*** $>$ ***rank (y)*** make ***x*** the parent of ***y*** else

if ***rank (x)*** $<$ ***rank (y)*** make ***y*** the parent of ***x*** else

if ***rank (x)*** $=$ ***rank (y)*** make ***x*** the parent of ***y*** and increase the rank of ***x*** by 1.

Use of either of the rules above improves the complexity drastically. In particular, the worst-case find time decreases from linear to logarithmic. Now the total cost for a sequence of m find operations and $n - 1$ intermixed unite operations is $(m \log n)$, because with either rule the depth of a tree is logarithmic in its size. This result (for union by size) is in [16].

There is one more thing we can do to improve the complexity of the solution to Problem 3.1. It is an idea that Knuth [29] attributes to Alan Tritter, and Hopcroft and Ullman [20] attribute to McIlroy and Morris. The idea is to modify the trees not only when we do ***unite*** operations, but also when we do ***find*** operations: when doing a ***find***, we "squash" the tree along the ***find*** path. (See Figure 4.) When we perform a ***find*** on an element, say ***E***, we walk up the path to the root, ***A***, which is the name of the set represented by this tree. We now know not only the answer for ***E***, but also the answer for every node along the path from ***E*** to the root. We take advantage of this fact by compressing this path, making all nodes on it point directly to the root. The tree is modified as depicted in Figure 4. Thus, if later we do a ***find*** on say, ***D***, this node is now one step away from it the root, instead of three steps away.

The question is, by how much does path compression improve the speed of the algorithm? Analyzing this algorithm, especially if both path compression and one of the ***unite*** rules is used, is complicated, and Knuth proposed it as a challenge. Note that if both path compression and union by rank are used, then the rank of tree root is not necessarily the tree height, but it is always an upper bound on the tree height. Let me remind you of the history of the bounds on this problem from the early 1970's.

There was an early incorrect "proof" of an $O(m)$ time-bound; that is, constant time per ***find***. Shortly thereafter, Mike Fischer [11] obtained a correct bound of

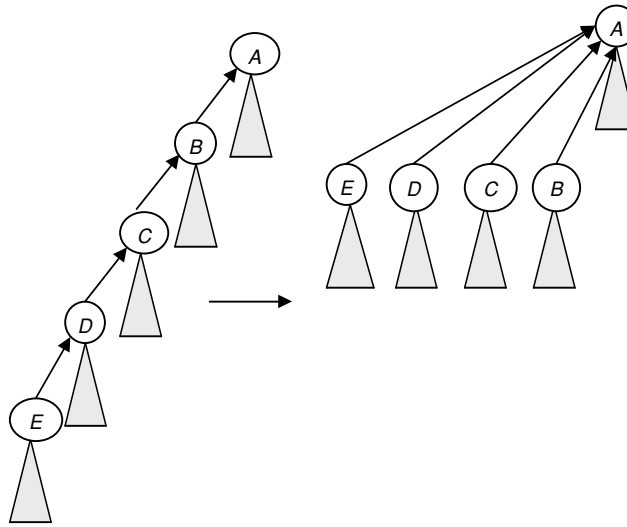


Figure 4.

$O(m \log\log n)$. Later, Hopcroft and Ullman [20] obtained the bound $O(m \log^* n)$. Here $\log^* n$ denotes the number of times one must apply the *log* function to n to get down to a constant. After this result had already appeared, there was yet another incorrect result, giving a lower bound of $\Omega(n \log\log n)$. Then I was able to obtain a lower bound which shows that this algorithm does not in fact perform in constant time per ***find***. Rather, its time per ***find*** is slightly worse than constant. Specifically, I showed a lower bound of $\Omega(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function, an incredibly slowly growing function that cannot possibly be measured in practice. It will be defined below. After obtaining the lower bound, I was able to get a matching upper bound of $O(m \cdot \alpha(n))$. (For both results, and some extensions, see [37,42].) So the correct answer for the complexity of the algorithm using both path compression and one of the ***unite*** rules is *almost constant* time per ***find***, where *almost constant* is the inverse of Ackermann's function.

Ackermann's function was originally constructed to be so rapidly growing that it is not in the class of primitively recursive functions, those definable by a single-variable recurrence. Here is a definition of the inverse of Ackermann's function. We define a sequence of functions:

$$\text{For } j \geq 1,\ k \geq 0,\ A_0(j) = j + 1, \quad A_k(j) = A_{k-1}^{(j+1)}(j) \text{ for } k \geq 1,$$

where $A^{(i+1)}(x) = A(A^{(i)}(x))$ denotes function composition.

Note that A_0 is just the successor function; A_1 is essentially multiplication by two, A_2 is exponentiation; A_3 is iterated exponentiation, the inverse of $\log^*(n)$; after that the functions grow very fast.

The inverse of Ackermann's function is defined as:

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

The growth of the function $\alpha(n)$ is incredibly slow. The smallest n such that $\alpha(n) = 4$ for example, is greater than any size of any problem that anyone will ever solve, in the course of all time.

The most interesting thing about this problem is the surprising emergence of $\alpha(n)$ in the time bound. I was able to obtain such a bound because I guessed that the truth was that the time per ***find*** is not constant (and I was right). Given this guess, I was able to construct a sequence of bad examples defined using a double recursion, which naturally led to Ackermann's function. Since I obtained this result, the inverse of Ackermann's function has turned up in a number of other places in computer science, especially in computational geometry, in bounding the complexity of various geometrical configurations involving lines and points and other objects [34].

Let one mention some further work on this problem. The tree data structure for representing sets, though simple, is very powerful. One can attach values to the edges or nodes of the trees and combine values along tree paths using ***find***. This idea has many applications [39]. There are variants of path compression that have the same inverse Ackermann function bound and some other variants that have worse bounds [42]. The lower bound that I originally obtained was for the particular algorithm that I have described. But the inverse Ackermann function turns out to be inherent in the problem. There is no way to solve the problem without having the inverse Ackermann function dependence. I was able to show this for a pointer machine computation model with certain restrictions [38]. Later Fredman and Saks [12] showed this for the cell probe computation model; theirs is a really beautiful result. Recently, Haim Kaplan, Nira Shafrir and I [24] have extended the data structure to support insertions and deletions of elements.

4. Amortization and Self-adjusting Search Trees

The analysis of path compression that leads to the inverse Ackermann function is complicated. But it illustrates a very important concept, which is the notion of *amortization*. The algorithm for Problem 3.1 performs a sequence of intermixed ***unite*** and ***find*** operations. Such operations can in fact produce a deep tree, causing at least one ***find*** operation to take logarithmic time. But such a ***find*** operation squashes the tree and causes later ***finds*** to be cheap. Since we are interested in measuring the total cost, we do not mind if some operations are expensive, as long as they are balanced by cheap ones. This leads to the notion of *amortizatized cost*, which is the cost per operation averaged over a worst-case sequence of operations. Problem 3.1 is the first example that I am aware of where this notion arose, although in the original work on the problem the word *amortization* was not used and the framework used nowadays for doing an amortized analysis was unknown then. The idea of a data structure in which simple

modifications improve things for later operations is extremely powerful. I would like to turn to another data structure in which this idea comes into play—*self-adjusting search trees.*

4.1. Search Trees

There is a type of self-adjusting search tree called the *splay* tree that I'm sure many of you know about. It was invented by Danny Sleater and me [35]. As we shall see, many complexity results are known for splay trees; these results rely on some clever ideas in algorithmic analysis. But the ultimate question of whether the splaying algorithm is optimal to within a constant factor remains an open problem.

Let me remind you about **binary search trees**.

Definition 4.1 *A **binary search tree** is a binary tree, (every node has a left and a right child, either of which, or both, can be missing.) Each node contains a distinct item of data. The items are selected from a totally ordered universe. The items are arranged in the binary search tree in the following way: for every node x in the tree, every node in the left subtree of x is less than the item stored in x and every node in the right subtree of x is greater than the item stored in x The operations done on the tree are **access**, **insert** and **delete**.*

We perform an *access* of an item in the obvious way: we start at the root, and we go down the tree, choosing at every node whether to go left or right by comparing the item in the node with the item we trying to find. The search time is proportional to the depth of the tree or, more precisely, the length of the path from the root to the designated item. For example, in Figure 5, a search for "frog", which is at the root, takes one step; a search for "zebra", takes four steps. Searching for "zebra" is more expensive, but not too expensive, because the tree is reasonably balanced. Of course, there are "bad" trees, such as long paths, and there are "good" trees, which are spread out wide like the one in Figure 5. If we have a fixed set of items, it is easy to construct a perfectly balanced tree, which gives us logarithmic worst-case access time.

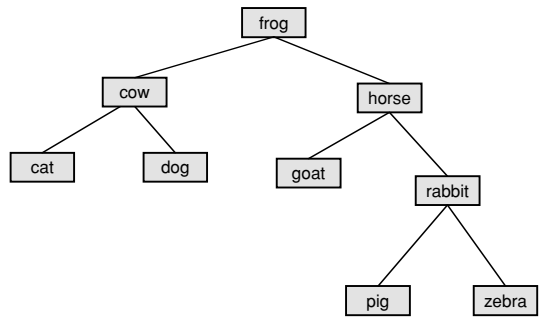


Figure 5.

The situation becomes more interesting if we want to allow ***insertion*** and ***deletion*** operations, since the shape of the tree will change. There are standard methods for inserting and deleting items in a binary search tree. Let me remind you how these work. The easiest method for an ***insert*** operation is just to follow the search path, which will run off the bottom of the tree, and put the new item in a new node attached where the search exits the tree. A ***delete*** operation is slightly more complicated. Consider the tree in Figure 5. If I want to delete say "pig" (a leaf in the tree in Figure 5), I simply delete the node containing it. But if I want to delete "frog", which is at the root, I have to replace that node with another node. I can get the replacement node by taking the left branch from the root and then going all the way down to the right, giving me the predecessor of "frog", which happens to be "dog", and moving it to replace the root. Or, symmetrically, I can take the successor of "frog" and move it to the position of "frog". In either case, the node used to replace frog has no children, so it can be moved without further changes to the tree. Such a replacement node can actually have one child (but not two); after moving such a node, we must replace it with its child. In any case, an insertion or deletion takes essentially one search in the tree plus a constant amount of restructuring. The time spent is at most proportional to the tree depth.

Insertion and deletion change the tree structure. Indeed, a bad sequence of such operations can create an unbalanced tree, in which accesses are expensive. To remedy this we need to restructure the tree somehow, to restore it to a "good" state.

The standard operation for restructuring trees is the rebalancing operation called ***rotation***. A ***rotation*** takes an edge such as (f, k) in the tree in Figure 6 and switches it around to become (k, f). The operation shown is a right rotation; the inverse operation is a left rotation. In a standard computer representation of a search tree, a rotation takes constant time; the resulting tree is still a binary search tree for the same set of ordered items. ***Rotation*** is universal in the sense that any tree on some set of ordered items can be turned into any other tree on the same set of ordered items by doing an appropriate sequence of rotations.

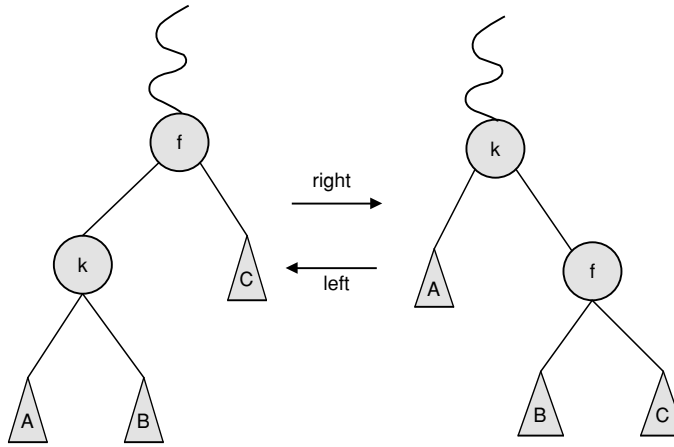


Figure 6.

We can use rotations to rebalance a tree when insertions and deletions occur. There are various "balanced tree" structures that use extra information to determine what rotations must be done to restore balance during insertions and deletions. Examples include **AVL trees** [1], **red-black trees** [19,40], and many others. All these balanced tree structures have the property that the worst-case time for a ***search***, ***insertion*** or ***deletion*** in an n-node tree is $O(\log n)$.

4.2. Splay Trees

This is not the end of the story, because balanced search trees have certain *drawbacks:*

- Extra space is needed to keep track of the balance information.
- The rebalancing needed for insertions and deletions involves several cases, and can be complicated.
- Perhaps most important, the data structure is logarithmic-worst-case but essentially logarithmic-best-case as well. The structure is not optimum for a non-uniform usage pattern. Suppose for example I have a tree with a million items but I only access a thousand of them. I would like the thousand items to be cheap to access, proportional to log (1,000), not to log (1,000,000). A standard balanced search tree does not provide this.

There are various data structures that have been invented to handle this last drawback. Assume that we know something about the usage pattern. For example, suppose we have an estimate of the access frequency for each item. Then we can construct an "optimum" search tree, which minimizes the average access time. But what if the access pattern changes over time? This happens often in practice.

Motivated by this issue and knowing about the amortized bound for path compression, Danny Sleator and I considered the following problem:

Problem 4.2 *Is there a simple, self-adjusting form of search tree that does not need an explicit balance condition but takes advantage of the usage pattern? That is, is there an update mechanism that adjusts the tree automatically, based on the way it is used?*

The goal is to have items that are accessed more frequently move up in the tree, and items that are accessed less frequently move down in the tree.

We were able to come up with such a structure, which we called the **splay tree.** A Splay tree is a self-adjusting search tree. See Figure 7 for an artist's conception of a self-adjusting tree.

"Splay" as a verb means to spread out. Splaying is a simple self-adjusting heuristic, like path compression, but that applies to binary search trees. The splaying heuristic takes a designated item and moves it up to the root of the tree by performing rotations,

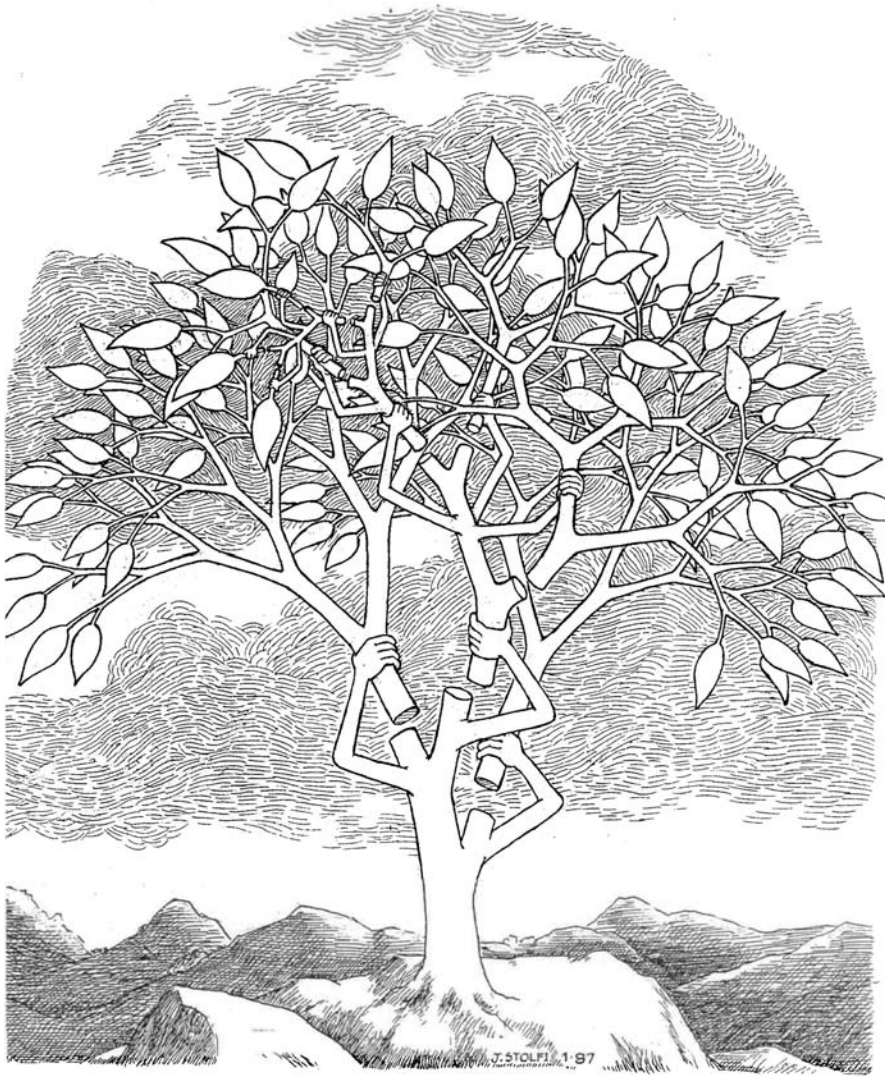


Figure 7.

preserving the item order. The starting point of the idea is to perform rotations bottom-up. It turns out, though, that doing rotations one at a time, in strict bottom-up order, does not produce good behavior. The splay heuristic performs rotations in pairs, in bottom-up order, according to the rules shown in Figure 8 and explained below. To access an item, we walk down the tree until reaching the item, and then perform the splay operation, which moves the item all the way up to the tree root. Every item along the search path has its distance to the root roughly halved, and all other nodes get pushed to the side. No node moves down more than a constant number of steps.

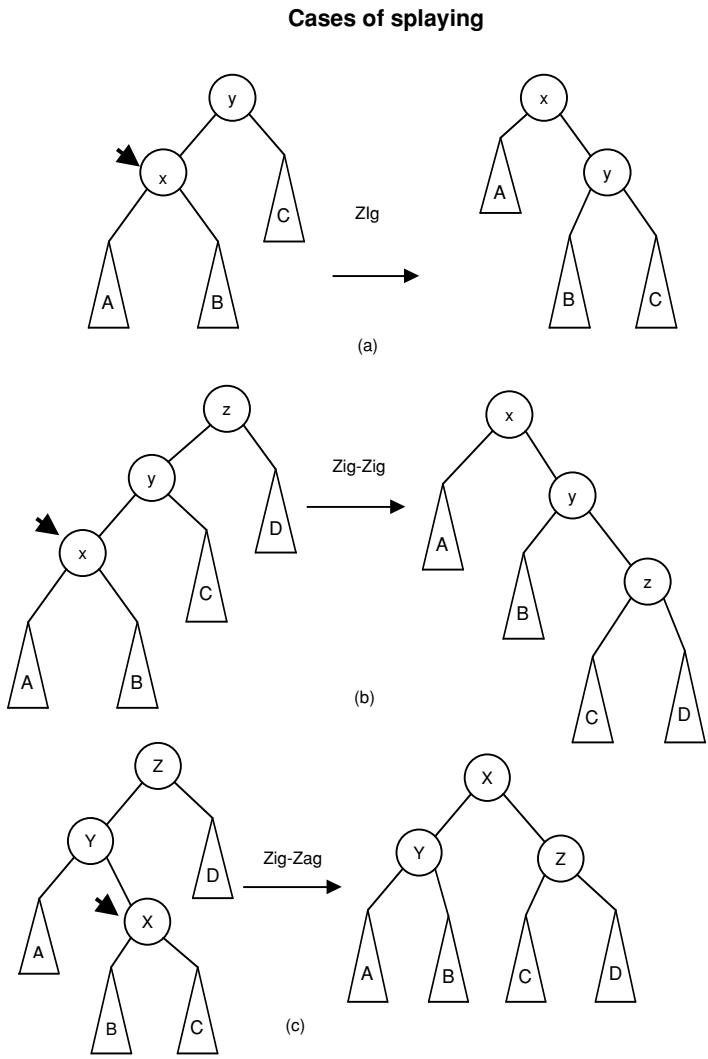


Figure 8.

Figure 8 shows the cases of a single splaying step. Assume x is the node to be accessed. If the two edges above x toward the root are in the same direction, we do a rotation on the top edge first and then on the bottom edge, which locally transforms the tree as seen in Figure 7b. This transformation doesn't look helpful; but in fact, when a sequence of such steps are performed, they have a positive effect. This is the "zig-zig" case. If the two edges from x toward the root are in opposite directions, such as right-left as seen in Figure 8c, or symmetrically left-right, then the bottom rotation is done first, followed by the top rotation. In this case, x moves up and y and z get split between the two subtrees of x. This is the zig-zag case. We keep doing zig-zag and

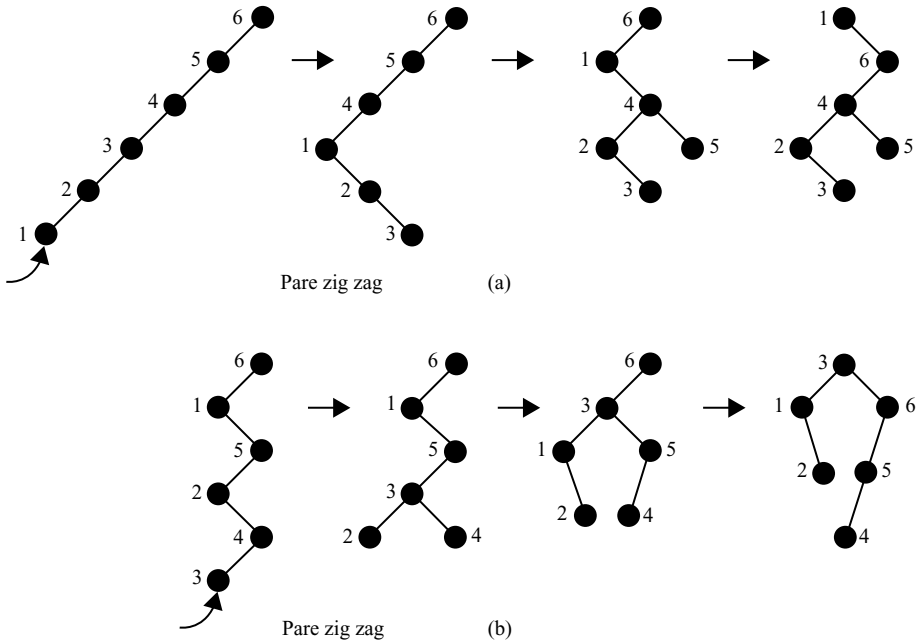


Figure 9.

zig-zig steps, as appropriate, moving x up two steps at a time, until either x is the root or it is one step away from the root. In the latter case we then do one final rotation, the zig case (Figure 8a). The *splay operation* is an entire sequence of splay steps that moves a designated item all the way up to the root.

Figure 9a contains a step-by-step example of a complete splay operation. This is a purely zig-zig case (except for a final zig). First we perform two rotations, moving item #1 up the path, and then two more rotations, moving item #1 further up the path. Finally, a last rotation is performed to make item #1 take root. Transforming the initial configuration into the final one is called "*splaying at node 1*". Figure 9b gives a step-by-step example of a purely zig-zag case.

Figure 10 is another example of a purely zig-zag case of a single splay operation. The accessed node moves to the root, every other node along the find path has its distance to the root roughly halved, and no nodes are pushed down by more than a constant amount. If we start out with a really bad example, and we do a number of splay operations, the tree gets balanced very quickly. The splay operation can be used not only during accesses but also as the basis of simple and efficient insertions, deletions, and other operations on search trees.

There is also a top-down version of splaying as well as other variants [35]. Sleator has posted code for the top-down version at [47].

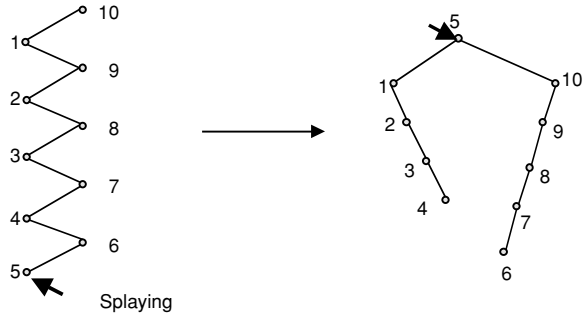


Figure 10.

4.3. Complexity Results

The main question for us as theoreticians is, "How does this algorithm perform?" Sleator and I were able to show that, in the amortized sense, this algorithm performs just as well as any balanced tree structure.

Theorem 4.3 *Beginning with an arbitrary n-node tree, suppose we perform a sequence of m* ***access****es and ignore start-up effects; that is, we assume that* $m \geq n$. *The following results hold:*

(a) The total cost of *m* ***access***es is $O(m \log n)$, thus matching the bound for balanced trees.
(b) The splaying algorithm on any access sequence performs within a constant factor of the performance of that of the best possible static tree for the given access sequence (in spite of the fact that this algorithm does not know the access frequencies ahead of time and doesn't keep track of them).
(c) If the items are accessed in increasing order, each once, the total access time is linear. That is, the amortized cost per access is constant, as compared to logarithmic for a balanced search tree or for any static tree. This result demonstrates that modifying the tree as operations proceed can dramatically improve the access cost in certain situations.

It is relatively straightforward to prove results (a) and (b) above. They follow from an interesting "potential" argument that I trust many of you know. These results can be found in [35], along with additional applications and extensions. Result (c) seems to be hard to prove — an elaborate inductive argument appears in [41].

The behavior of a splay tree when sequential accesses are performed is quite interesting. Suppose we start with an extreme tree that is just a long left path, with the smallest item at the bottom of the path, and begin accessing the items in increasing order. The first access costs n. The next access costs about $n/2$, the next one costs $n/4$, and so on. The access time drops by about a factor of two with each access until after about logarithmically many accesses, at which time the tree is quite well-balanced.

Then the time per access starts behaving something like a ruler function, with a certain amount of randomness thrown in: about half the accesses take constant time; about a quarter take about twice as long; about an eighth take about three times as long; and so on. The amortized cost per access is constant rather than logarithmic.

Based on results (a)—(c) and others, Sleator and I made (what we consider to be) an audacious conjecture. Suppose we begin with some n-node search tree and perform a sequence of accesses. (For simplicity, we ignore insertions and deletions.) The cost to access an item is the number of nodes on the path from the root to the item. At any time, we can perform one or more rotations, at a cost of one per rotation. Then, knowing the entire access sequence ahead of time, there is a pattern of rotations that minimizes the total cost of the accesses and the rotations. This is the optimal offline algorithm, for the given initial tree and the given access sequence.

Problem 4.4 ***Dynamic Optimality Conjecture:*** *Prove or disprove that for any initial tree and any access sequence, the splaying algorithm comes within a constant factor of the optimal off-line algorithm.*

Note that splaying is an on-line algorithm: its behavior is independent of future accesses, depending only on the initial tree and the accesses done so far. Furthermore the only information the algorithm retains about the accesses done so far is implicit in the current state of the tree. A weaker form of Problem 4.4 asks whether there is *any* on-line algorithm whose performance is within a constant factor of that of the optimum off-line algorithm.

The closest that anyone has come to proving the strong form of the conjecture is an extension by Richard Cole and colleagues of the sequential access result, Theorem 4.3(c). Consider a sequence of accesses. For simplicity, assume that the items are 1, 2, 3, in the corresponding order. The distance between two items, say i and j, is defined to be $|i - j| + 2$. Cole et al. proved the following:

Theorem 4.4 *[6,7]: The total cost of a sequence of accesses using splaying is at most of the order of n plus m plus the sum of the logarithms of the distances between consecutively accessed items.*

This result, the "dynamic finger theorem", is a special case of the (strong) dynamic optimality conjecture. Significant progress on the weak form of the conjecture has been made recently by Demaine and colleagues [9]. They have designed an on-line search tree update algorithm that comes within a *loglog* performance factor of the optimal off-line algorithm (as compared to the log factor of a balanced tree algorithm). A key part of their result is a lower bound on the cost of an access sequence derived by Bob Wilber [43]. The algorithm of Demaine et al. is cleverly constructed so that its performance comes within a loglog factor of Wilber's lower bound. They also argue that use of Wilber's lower bound will offer no improvement beyond loglog. An immediate question is whether one can prove a similar loglog bound for splaying, even if the dynamic optimality conjecture remains out of reach.

To summarize, we do not know the answer to the following question:

Is the splaying algorithm (or any on-line binary search tree algorithm) optimal to within a constant factor?

Splaying as well as path compression are examples of very simple operations that give rise when repeated to complicated, hard-to-analyze, behavior. Splay trees have been used in various systems applications, for memory management, table storage, and other things. In many applications of tables, most of the data does not get accessed most of the time. The splaying tree algorithm takes advantage of such locality of reference: the splay algorithm moves the current working set to the top of the tree, where it is accessible at low cost. As the working set changes, the new working set moves to the top of the tree.

The drawback of this data structure is, of course, that it performs rotations all the time, during accesses as well as during insertions and deletions. Nevertheless, it seems to work very well in practical situations in which the access pattern is changing.

5. The Rotation Distance between Search Trees

Search trees are a fascinating topic, not just because they are useful data structures, but also because they have interesting mathematical properties. I have mentioned that any search tree can be transformed into any other search tree on the same set of ordered items by performing an appropriate sequence of rotations. This fact raises at least two interesting questions, one algorithmic, one structural.

Problem 5.1 *Given two n-node trees, how many rotations does it take to convert one tree into the other?*

This question seems to be NP- hard. A related question is how far apart can two trees be? To formulate the question more precisely, we define the *rotation graph* on n-node trees in the following way: The vertices of the graph are the n-node binary trees. Two trees are connected by an edge if and only if one tree can be obtained from the other by doing a single rotation. This graph is connected.

Problem 5.2 *What is the diameter of the rotation graph described above as a function of n?*

It is easy to get an upper bound of $2n$ for the diameter. It is also easy to get a lower bound of $n - O(1)$, but this leaves a factor of two gap. Sleator and I worked with Bill Thurston on this problem. He somehow manages to map every possible problem into hyperbolic geometry. Using mostly Thurston's ideas, we were able to show that the $2n$ bound is tight, by applying a volumetric argument in hyperbolic space. We were even able to establish the exact bound for large enough n, which is $2n - 6$ [36]. To me this is an amazing result. The details get quite technical, but this is an example in which a

piece of mathematics gets brought in from way out in left field to solve a nitty-gritty data structure problem. This just goes to show the power of mathematics in the world.

6. Static Optimum Search Trees

A final question about search trees concerns static optimum trees. Suppose we want to store n ordered items in a fixed search tree so as to minimize the average access time, assuming that each item is accessed independently with a fixed probability, and that we know the access probabilities. The problem is to construct the best tree. There are an exponential number of trees, so it might take exponential time find the best one, but in fact it does not. There are two versions of this problem, depending on the kind of binary search tree we want.

Problem 6.1 *Items can be stored in the internal nodes of the tree. (This is the kind of search tree discussed in Sections 4 and 5.)*

The problem in this case is: Given positive weights $w_1, w_2, \ldots, w_n$, construct an n-node binary tree that minimizes $\sum_{i=1}^{n} w_i d_i$, where d_i is the depth of the i-th node in symmetric order.

Problem 6.2 *Items can be stored only in the external nodes (the leaves) of the tree.*

The problem in this case is: Given positive weights $w_1, w_2, \ldots, w_n$, construct a binary tree with n external nodes that minimizes $\sum_{i=1}^{n} w_i d_i$, where d_i is the depth of the i-th external node in symmetric order.

Problem 6.1 can be solved by a straightforward $O(n^3)$-time dynamic programming algorithm (as in the stack generation problem discussed in Section 2). Knuth[30] was able to improve on this by showing that there is no need to look at all the subproblems; there is a restriction on the subproblems that have to be considered, which reduces the time to $O(n^2)$. This result was extended by Frances Yao [45] to other problems. Yao captured a certain kind of structural restriction that she called "quadrangle inequalities". Here we have an $O(n^2)$ dynamic programming algorithm with a certain amount of cleverness in it. This result is twenty-five years old or so. Nothing better is known. There may, in fact, be a faster algorithm for Problem 6.1; no nontrivial lower bound for the problem is known.

For Problem 6.2, in which the items are stored in the external nodes and the internal nodes just contain values used to support searching, we can do better. There is a beautiful algorithm due to Hu and Tucker [21] that runs in $O(n \log n)$ time. This algorithm is an extension of the classical algorithm of Huffman [22] for building so-called "Huffman codes". The difference between Hoffman codes and search trees is the alphabetic restriction: in a search tree the items have to be in a particular order in the leaves, whereas in a Huffman code they can be permuted arbitrarily. It turns out that

a simple variant of the Huffman coding algorithm solves Problem 6.2. The amazing thing about this is that the proof of correctness is inordinately complicated. Garsia and Wachs [17] presented a variant of the algorithm with an arguably simpler proof of correctness, and Karpinski et al. [26] have made further progress on simplifying the proofs of both algorithms. But it is still quite a miracle that these simple algorithms work. And again, there is no lower bound known. In fact, there is no reason to believe that the problem cannot be solved in linear time, and there is good reason to believe that maybe it can: for some interesting special cases the problem can indeed be solved in linear time [27].

7. The Minimum Spanning Tree Problem

Let me close by coming back to a classical graph problem—the minimum spanning tree problem. Given a connected, undirected graph with edge costs, we want to find a spanning tree of minimum total edge cost. This problem is a classical network optimization problem, and it is about the simplest problem in this area. It has a very long history, nicely described by Graham and Hell [18]. The first fully-realized algorithms for this problem were developed in the 1920's. There are three classic algorithms. The most recently discovered is Kruskal's algorithm [31], with which we are all familiar. This algorithm processes the edges in increasing order by cost, building up the tree edge-by-edge. If a given edge connects two different connected components of what has been built so far, we add it as an additional edge; if it connects two vertices in the same component, we throw it away. To implement this algorithm, we need an algorithm to sort (or at least partially sort) the edges, plus a data structure to keep track of the components. Keeping track of the components is exactly the set union problem discussed in Section 3. The running time of Kruskal's algorithm is $O(m \log n)$ including the sorting time. If the edges are presorted by cost, the running time of the algorithm is the same as that of disjoint set union; namely, $O(m\,\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

An earlier algorithm, usually credited to Prim [33] and Dijkstra [10], is a single-source version of Kruskal's algorithm. It begins with a vertex and grows a tree from it by repeatedly adding the cheapest edge connecting the tree to a new vertex. This algorithm was actually discovered by Jarník in 1930 [23]. It runs in $O(n^2)$ time, as implemented by Prim and Dijkstra. (Jarník did not consider its computational complexity at all.) By using a data structure called a heap (or priority queue) the algorithm can be implemented to run in $O(m \log n)$ time.

An even earlier algorithm, a beautiful parallel one, was described by Boruvka [3] in 1926. It also has a running time of $O(m \log n)$. (The time bound is recent; Boruvka did not investigate its complexity.) This algorithm is Kruskal's algorithm in parallel. In the first iteration, for every vertex pick the cheapest incident edge and add it the set of edges so far selected. In general, the chosen edges form a set of trees. As long as there are at least two such trees, pick the cheapest edge incident to each tree and add these

edges to the chosen set. In every iteration the number of trees decreases by at least a factor of two, giving a logarithmic bound on the number of iterations. If there are edges of equal cost, a consistent tie-breaking rule must be used, or the algorithm will create cycles, but this is just a technical detail.

All three classic algorithms run in $O(m \log n)$ time if they are implemented appropriately. The obvious question is whether sorting is really needed, or can one can get rid of the log factor and solve this problem in linear time? Andy Yao [44] was the first to make progress. He added a new idea to Boruvka's algorithm that reduced the running time to $O(m \log \log n)$. He thereby showed that sorting was not, in fact, inherent in the minimum spanning tree problem. A sequence of improvements followed, all based on Boruvka's algorithm. Mike Fredman and I [13] achieved a running time of $O(m \log^* n)$ by using a new data structure called the *Fibonacci heap* in combination with Boruvka's algorithm. By adding a third idea, that of *packets*, Galil, Gabor, and Spencer [14] (see also [15]) obtained $O(m \log \log^* n)$ running time. Finally, Klein and Tarjan [28] (see also [25]) used random sampling in combination with a linear-time algorithm for verification of minimum spanning trees to reach the ultimate goal, a linear time algorithm for finding minimum spanning trees.

One may ask whether random sampling is really necessary. Is there a deterministic linear-time algorithm to find minimum spanning trees? Bernard Chazelle [4,5], in a remarkable tour de force, invented a new data structure, called the **soft heap**, and used it to obtain a minimum spanning tree algorithm with a running time of $O(m \alpha(n))$. (Our old friend, the inverse Ackermann function, shows up again!) Soft heaps make errors, but only in a controlled way. This idea of allowing controlled errors is reminiscent of the remarkable (and remarkably complicated) $O(n \log n)$-size sorting network of Aijtai, Komlos, and Szemerédi [2], which uses partition sorting with careful control of errors.

Chazelle's ideas were used in a different way by Pettie and Ramachandran [32] who designed a minimum spanning tree algorithm that has a running time optimum to within a constant factor, but whose running time they could not actually analyze. The extra idea in their construction is that of building an optimum algorithm for very-small-size subproblems by exhaustively searching the space of all possible algorithms, and then combining this algorithm with a fixed-depth recursion based on Chazelle's approach.

In conclusion, we now know that minimum spanning trees can be found in linear time using random sampling, and we know an optimum deterministic algorithm but we don't know how fast it runs (no slower than $O(m \alpha(n))$, possibly as fast as $O(m)$). This is the strange unsettled state of the minimum spanning tree problem at the moment.

Acknowledgments

My thanks to Mitra Kelly, Sandra Barreto, and Irith Hartman for their hard work in helping prepare the manuscript.

References

[1] G. M. Adel'son-Vel'skii and E.M. Landis, An algorithm for the organization of information, *Soviet Math Dokl*. 3: 1259–1262 (1962).

[2] M. Ajtai, J. Komlós, and E. Szemerédi, Sorting in $c \log n$ parallel steps, *Combinatorica* 3: 1–19 (1983).

[3] O. Borůvka, O jistém problému minimálním, *Práce Mor. Přírověd. Spol. v Brně (Acta Societ. Scient. Natur. Moravicae)* 3: 37–58 (1926).

[4] B. Chazelle, The soft heap: an approximate priority queue with optimal error rate, *J. Assoc. Comput. Mach*. 47: 1012–1027 (2000).

[5] B. Chazelle, A minimum spanning tree algorithm with inverse-Ackermann type complexity, *J. Assoc. Comput. Mach*. 47: 1028–1047 (2000).

[6] R. Cole, B. Mishra, J. Schmidt, and A. Siegel, On the dynamic finger conjecture for splay trees, part I: splay sorting log$n$ -block sequences, *SIAM J. Computing* 30: 1–43 (2000).

[7] R. Cole, On the dynamic finger conjecture for splay trees, part II: the proof, *SIAM J. Computing* 30: 44–85 (2000).

[8] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Computation* 9: 251–280 (1990).

[9] E. D. Demaine, D. Harmon, J. Iacono, and M. Pătraşcu, Dynamic optimality—almost, *Proc 45th Annual IEEE Symp. on Foundations of Computer Science,* (2004) pp. 484–490.

[10] E. W. Dijkstra, A note on two problems in connexion with graphs, *Num. Mathematik* 1: 269–271 (1959).

[11] M. J. Fischer, Efficiency of equivalence algorithms, *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, NY, (1972) pp. 153–168.

[12] M. L. Fredman and M. E. Saks, The cell probe complexity of dynamic data structures, *Proc. 21st Annual ACM Symp. on Theory of Computing*, (1989) pp. 345–354.

[13] M. R. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach*. 34: 596–615 (1987).

[14] H. N. Gabow, Z. Galil, and T. H. Spencer, Efficient implementation of graph algorithms using contraction, *Proc. 25th Annual IEEE Symp. on Found. of Comp. Sci.,* (1984) pp. 347–357.

[15] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in directed and undirected graphs, *Combinatorica* 6: 109–122 (1986).

[16] B. A. Galler and M. J. Fischer, An improved equivalence algorithm, *Comm. Assoc. Comput. Mach*. 7: 301–303 (1964).

[17] A. M. Garsia and M. L. Wachs, A new algorithm for minimal binary encodings, *SIAM J. Comput*. 6: 622–642 (1977).

[18] R. L. Graham and P. Hell, On the history of the minimum spanning tree problem, *Annuals of the History of Computing* 7: 43–57 (1985).

[19] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *Proc. 19th Annual IEEE Symp. on Foundations of Computer Science,* (1978) pp. 8–21.

[20] J. Hopcroft and J. D. Ullman, Set-merging algorithms, *SIAM J. Comput*. 2: 294–303 (1973).

[21] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable-length alphabetic codes, *SIAM J. Appl. Math* 21: 514–532 (1971).

[22] D. A. Huffman, A method for the construction of minimum-redundancy codes, *Proc. IRE* 40: 1098–1101 (1952).

[23] V. Jarník, O jistém problému minimálním, *Práce Mor. Přírověd. Spol. v Brně (Acta Societ. Scient. Natur. Moraricae)* 6: 57–63 (1930).

[24] H. Kaplan, N. Shafrir, and R. E. Tarjan, Union-find with deletions, *Proc. 13th Annual ACM-SIAM Symp. on Discrete Algorithms,* (2002) pp. 19–28.

[25] D. R. Karger, P. N. Klein, and R. E. Tarjan, A randomized linear-time algorithm to find minimum spanning trees, *J. Assoc. Comput. Mach*. 42: 321–328 (1995).

[26] M. Karpinski, L. L. Larmore, and V. Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science* 180: 309–324 (1997).

[27] M. Klawe and B. Mumey, Upper and lower bounds on constructing alphabetic binary trees, *Proc 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, (1993) pp. 185–193.

[28] P. N. Klein and R. E. Tarjan, A randomized linear-time algorithm for finding minimum spanning trees, *Proc. 26th Annual ACM Symp. on Theory of Computing*, (1994) pp. 9–15.

[29] D. E. Knuth, The *Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, (1973) p. 572.

[30] D. E. Knuth, Optimum binary search trees, *Acta Informatica* 1: 14–25 (1971).

[31] J. B. Kruskal, On the shortest spannng subtree of a graph and the travelling saleman problem, *Proc. Amer. Math. Soc.* 7: 48–50 (1956).

[32] S. Pettie and V. Ramachandran, An optimal minimum spanning tree algorithm, *J. Assoc. Comput. Mach.* 49: 16–34 (2002).

[33] R. C. Prim, The shortest connecting network and some generalizations, *Bell Syst. Tech. J.* 36: 1389–1401 (1957).

[34] M. Sharir and P. K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge, England, (1995).

[35] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. Assoc. Comput. Mach*. 32: 652–686 (1985).

[36] D. D. Sleator, R. E. Tarjan, and W. P. Thurston, Rotation distance, triangulations, and hyperbolic geometry, *J. Amer. Math Soc*. 1: 647–682 (1988).

[37] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach*. 22: 215–225 (1975).

[38] R. E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. Syst. Sci*. 18: 110–127 (1979).

[39] R. E. Tarjan, Applications of path compression on balanced trees, *J. Assoc. Comput. Mach*. 26: 690–715 (1979).

[40] R. E. Tarjan, Updating a balanced search tree in O(1) rotations, *Info. Process. Lett*. 16: 253–257 (1983).

[41] R. E. Tarjan, Sequential access in splay trees takes linear time, *Combinatorica* 5: 367–378 (1985).

[42] R. E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.* 31: 246–281 (1984).

[43] R. Wilbur, Lower bounds for accessing binary search trees with rotations, *SIAM J. Computing* 18: 56–67 (1989).

[44] A. C. Yao, An O(|E|loglog|V|) algorithm for finding minimum spanning trees, *Info. Process. Lett*. 4: 21–23 (1975).

[45] F. F. Yao, Efficient dynamic programming using quadrangle inequalities, *Proc. 12th Annual ACM Symp. on Theory of Computing*, (1980) pp. 429–435.

[46] http://cristal.inria.fr/ICFP2001/prog-contest/

[47] http://www.link.cs.cmu.edu/splay/

3

Algorithmic Graph Theory and Its Applications*

Martin Charles Golumbic
Caesarea Rothschild Institute
University of Haifa
Haifa, Israel

1. Introduction

The topic about which I will be speaking, algorithmic graph theory, is part of the interface between combinatorial mathematics and computer science. I will begin by explaining and motivating the concept of an intersection graph, and I will provide examples of how they are useful for applications in computation, operations research, and even molecular biology. We will see graph coloring algorithms being used for scheduling classrooms or airplanes, allocating machines or personnel to jobs, or designing circuits. Rich mathematical problems also arise in the study of intersection graphs, and a spectrum of research results, both simple and sophisticated, will be presented. At the end, I will provide a number of references for further reading.

I would like to start by defining some of my terms. For those of you who are professional graph theorists, you will just have to sit back and enjoy the view. For those of you who are not, you will be able to learn something about the subject. We have a mixed crowd in the audience: university students, high school teachers, interdisciplinary researchers and professors who are specialists in this area. I am gearing this talk so that it will be non-technical, so everyone should be able to enjoy something from it. Even when we move to advanced topics, I will not abandon the novice.

When I talk about a graph, I will be talking about a collection of vertices and edges connecting them, as illustrated in Figures 1 and 2. Graphs can be used in lots of different applications, and there are many deep theories that involve using graphs. Consider, for example, how cities may be connected by roads or flights, or how documents might

* This chapter is based on the Andrew F. Sobczyk Memorial Lecture delivered by the author on October 23, 2003 at Clemson University. For a biographical sketch of Andrew F. Sobczyk, see the website http://www.math.clemson.edu/history/sobczyk.html

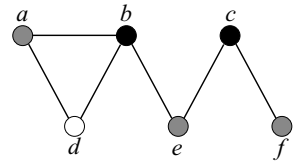


Figure 1. A graph and a coloring of its vertices.

be connected by similar words and topics. These are regularly modeled by graphs, and actions on them are carried out by algorithms applied to these graphs. Some of the terms that we need are:

- *Coloring a graph* – coloring a graph means assigning a color to every vertex, with the property that two vertices that are adjacent, i.e., connected by an edge, have different colors. As you can see in the example in Figure 1, we have colored the vertices white, grey or black. Notice that whenever a pair of vertices are joined by an edge, they have different colors. For example, a black vertex can be connected *only* to grey or white vertices. It is certainly possible to find pairs that have different colors yet are not connected, but every time we have an edge, its two end points must be different colors. That is what we mean by coloring.
- An *independent set* or a *stable set* – a collection of vertices, no two of which are connected. For example, in Figure 1, the grey vertices are pair-wise not connected, so they are an independent set. The set of vertices $\{d, e, f\}$ is also an independent set.
- A *clique* or a *complete* subset of vertices – a collection of vertices where everything is connected to each other, i.e., every two vertices in a clique are connected by an edge. In our example, the vertices of the triangle form a clique of size three. An edge is also a clique – it is a small one!
- The *complement* of a graph – when we have a graph we can turn it "inside out" by turning the edges into non-edges and vice-versa, non-edges into edges. In this way, we obtain what is called the complement of the graph, simply interchanging the edges and the non-edges. For example, the complement of the graph in Figure 1 is shown in Figure 3. We denote the complement of G by $\overline{G}$.
- An *orientation* of a graph – an orientation of a graph is obtained by giving a direction to each edge, analogous to making all the streets one way. There are many different ways to do this, since every edge could go either one way or the other. There are names for a number of special kinds of orientations. Looking at Figure 2, the first orientation of the pentagon is called *cyclic*, its edges are

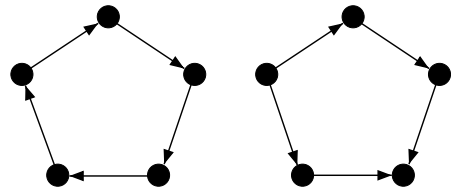


Figure 2. Two oriented graphs. The first is cyclic while the second is acyclic.

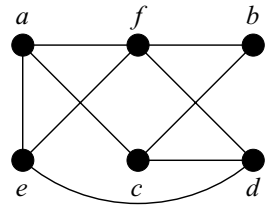


Figure 3. The complement of the graph in Figure 1.

directed consistently around in a circle, in this case clockwise; one can go round and round cycling around the orientation forever. The opposite of this idea, when there is no oriented cycle, is called an *acyclic* orientation. The second orientation of the pentagon is an acyclic orientation. You cannot go around and around in circles on this – wherever you start, it keeps heading you in one direction with no possibility of returning to your starting point.

Another kind of orientation is called a transitive orientation. An orientation is *transitive* if every path of length two has a "shortcut" of length one.[1] In Figure 4, the first orientation is not transitive because we could go from vertex d to vertex b and then over to vertex c, without having a shortcut. This is not a transitive orientation. The second orientation is transitive because for every triple of vertices x, y, z, whenever we have an edge oriented from x to y and another from y to z, then there is always a shortcut straight from x to z. Not all graphs have an orientation like this. For example, the pentagon cannot possibly be oriented in a transitive manner since it is a cycle of odd length.

2. Motivation: Interval Graphs

2.1. An Example

Let us look now at the motivation for one of the problems I will discuss. Suppose we have some lectures that are supposed to be scheduled at the university, meeting at certain hours of the day. Lecture a starts at 09:00 in the morning and finishes at 10:15; lecture b starts at 10:00 and goes until 12:00 and so forth. We can depict this on the real line by intervals, as in Figure 5. Some of these intervals intersect, for example, lectures a and b intersect from 10:00 until 10:15, the period of time when they are both in session. There is a point in time, in fact, where four lectures are "active" at the same time.

We are particularly interested in the intersection of intervals. The classical model that we are going to be studying is called an *interval graph* or the *intersection graph of a collection of intervals*. For each of the lectures, we draw a vertex of the interval graph, and we join a pair of vertices by an edge if their intervals intersect. In our

[1] Formally, if there are oriented edges $x \to y$ and $y \to z$, then there must be an oriented edge $x \to z$.

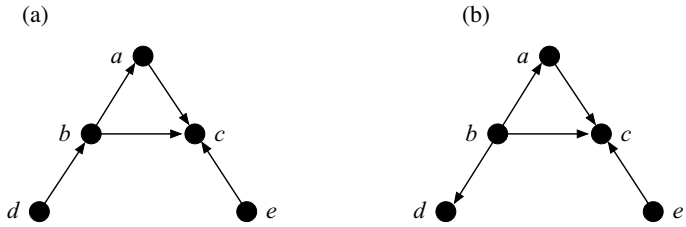


Figure 4. A graph with two orientations. The first is not transitive while the second is transitive.

example, lectures a and b intersect, so we put an edge between vertex a and vertex b, see Figure 6(a). The same can be done for lectures b and c since they intersect, and so forth. At time 13:15, illustrated by the vertical cut, we have c, d, e and f all intersecting at the same time, and sure enough, they have edges between them in the graph. They even form a clique, a complete subset of vertices, because they pair-wise intersect with each other. Some of the intervals are disjoint. For example, lecture a is disjoint from lecture d, so there is no edge between vertices a and d.

Formally, a graph is an *interval graph* if it is the intersection graph of some collection of intervals[2] on the real line.

Those pairs of intervals that do not intersect are called disjoint. It is not surprising that if you were to consider a graph whose edges correspond to the pairs of intervals that are disjoint from one another, you would get the complement of the intersection graph, which we call the *disjointness graph*. It also happens that since these are intervals on the line, when two intervals are disjoint, one of them is before the other. In this case, we can assign an orientation on the (disjointness) edge to show which interval is earlier and which is later. See Figure 6(b). Mathematically, this orientation is a partial order, and as a graph it is a transitive orientation. If there happens to be a student here from Professor Jamison's Discrete Math course earlier today, where he taught about Hasse diagrams, she will notice that Figure 6(b) is a Hasse diagram for our example.

2.2. Good News and Bad News

What can we say about intersecting objects? There is both good and bad news. Intersection can be regarded as a good thing, for example, when there is something important in common between the intersecting objects – you can then share this commonality, which we visualize mathematically as covering problems. For example, suppose I want to make an announcement over the loudspeaker system in the whole university for everyone to hear. If I pick a good time to make this public announcement, all the classes that are in session (intersecting) at that particular time will hear the announcement. This might be a good instance of intersection.

[2] The intervals may be either closed intervals which include their endpoints, or open intervals which do not. In this example, the intervals are closed.

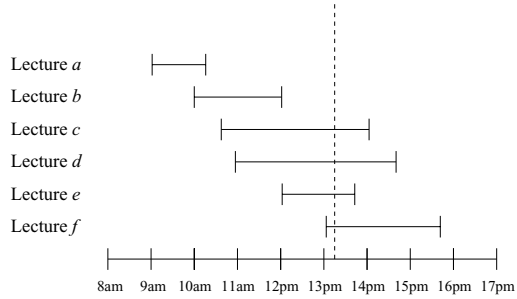


Figure 5. An interval representation.

Intersection can also be a bad thing, such as when intersecting intervals indicate a conflict or competition, and the resource cannot be shared. In our example of scheduling university lectures, we cannot put two lectures in the same classroom if they are meeting at the same time, thus, they would need different classes. Problems such as these, where the intervals cannot share the same resource, we visualize mathematically as coloring problems and maximum independent set problems.

2.3. Interval Graphs and their Applications

As mentioned earlier, not every graph can be an interval graph. The problem of characterizing which graphs could be interval graphs goes back to the Hungarian mathematician Gyorgy Hajös in 1957, and independently to the American biologist, Seymour Benzer in 1959. Hajös posed the question in the context of overlapping time intervals, whereas Benzer was looking at the linear structure of genetic material, what we call genes today. Specifically, Benzer asked whether the sub-elements could be arranged in a linear arrangement. Their original statements of the problem are quoted in [1] page 171. I will have more to say about the biological application later.

We have already seen the application of scheduling rooms for lectures. Of course, the intervals could also represent meetings at the Congress where we may need to

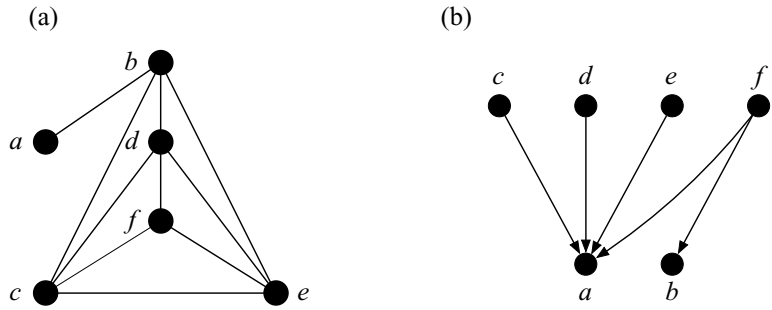


Figure 6. (a) The interval graph of the interval representation in Figure 5 and (b) a transitive orientation of its complement.

allocate TV crews to each of the meetings. Or there could be applications in which jobs are to be processed according to a given time schedule, with concurrent jobs needing different machines. Similarly, there could be taxicabs that have to shuttle people according to a fixed schedule of trips. The assignment problem common to all these applications, classrooms to courses, machines to jobs, taxis to trips, and so on, is to obtain a feasible solution – one in which no two courses may meet in the same room at the same time, and every machine or taxi does one job at a time.

3. Coloring Interval Graphs

The solution to this problem, in graph theoretic terms, is to find a coloring of the vertices of the interval graph. Each color could be thought of as being a different room, and each course needs to have a room: if two classes conflict, they have to get two different rooms, say, the brown one and the red one. We may be interested in a feasible coloring or a minimum coloring – a coloring that gives the fewest number of possible classrooms.

Those who are familiar with algorithms know that some problems are hard and some of them are not so hard, and that the graph coloring problem "in general" happens to be one of those hard problems. If I am given a graph with a thousand vertices with the task of finding a minimum feasible coloring, i.e., a coloring with the smallest possible number of colors, I will have to spend a lot of computing time to find an optimal solution. It could take several weeks or months. The coloring problem is an NP-complete problem, which means that, in general, it is a difficult, computationally hard problem, potentially needing an exponentially long period of time to solve optimally.

However, there is good news in that we are not talking about any kind of graph. We are talking about interval graphs, and interval graphs have special properties. We can take advantage of these properties in order to color them efficiently. I am going to show you how to do this on an example.

Suppose we have a set of intervals, as in Figure 7. You might be given the intervals as pairs of endpoints, [1, 6], [2, 4], [3, 11] and so forth, or in some other format like a sorted list of the endpoints shown in Figure 8. Figure 7 also shows the interval graph. Now we can go ahead and try to color it. The coloring algorithm uses the nice diagram of the intervals in Figure 8, where the intervals are sorted by their left endpoints, and this is the order in which they are processed. The coloring algorithm sweeps across from left to right assigning colors in what we call a "greedy manner". Interval a is the first to start – we will give it a color, solid "black". We come to b and give it the color "dashes", and now we come to c and give it the color "dots". Continuing across the diagram, notice "dashes" has finished. Now we have a little bit of time and d starts. I can give it "dashes" again. Next "black" becomes free so I give the next interval, e, the color "black". Now I am at a trouble spot because "dots", "dashes" and "black" are all busy. So I have to open up a new color called "brown" and assign that color

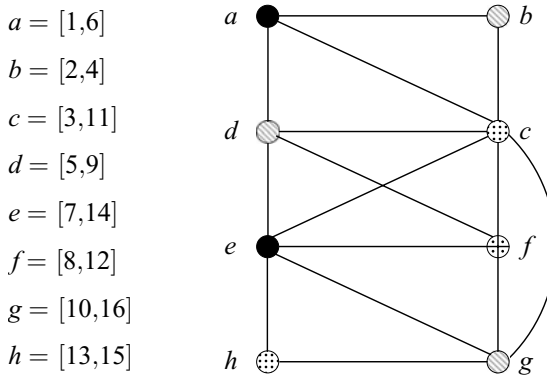


Figure 7. A set of intervals and the corresponding (colored) interval graph.

to interval f. I continue coloring from left to right and finally finish at the end. This greedy method gives us a coloring using 4 colors.

Is it the best we can do? Mathematicians would ask that question. Can you "prove" that this is the best we can do? Can we show that the greedy method gives the smallest possible number of colors? The answer to these questions is "yes".

Since this is a mathematics lecture, we must have a proof. Indeed, the greedy method of coloring is optimal, and here is a very simple proof. Let k be the number of colors that the algorithm used. Now let's look at the point P, as we sweep across the intervals, when color k was used for the first time. In our example, $k = 4$ and $P = 8$ (the point when we had to open up the color "brown".) When we look at the point P, we observe that all the colors 1 through $k - 1$ were busy, which is why we had to open up the last color k. How many intervals (lectures) are alive and running at that point P?

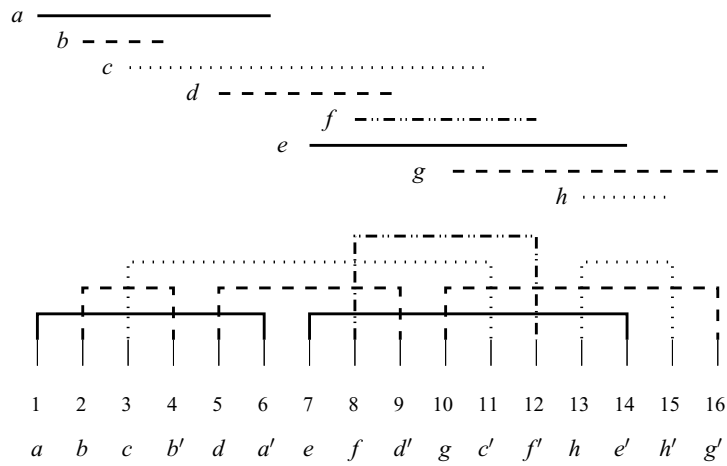


Figure 8. A sorted list of endpoints of the intervals in Figure 7.

The answer is k. I am forced to use k colors, and in the interval graph, they form a clique of size k. Formally, (1) the intervals crossing point P demonstrate that there is a k-clique in the interval graph – which means that at least k colors are required in any possible coloring, and (2) the greedy algorithm succeeded in coloring the graph using k colors. Therefore, the solution is optimal. Q.E.D.

It would be nice if all theorems had simple short proofs like this. Luckily, all the ones in this lecture will.

Interval graphs have become quite important because of their many applications. They started off in genetics and in scheduling, as we mentioned earlier. They have applications in what is called seriation, in archeology and in artificial intelligence and temporal reasoning. They have applications in mobile radio frequency assignment, computer storage and VLSI design. For those who are interested in reading more in this area, several good books are available and referenced at the end.

4. Characterizing Interval Graphs

What are the properties of interval graphs that may allow one to recognize them? What is their mathematical structure? I told you that not all graphs are interval graphs, which you may have believed. Now I am going to show you that this is true. There are two properties which together characterize interval graphs; one is the chordal graph property and the other is the co-TRO property.

A graph is *chordal* if every cycle of length greater than or equal to four has a chord. A chord means a diagonal, an edge that connects two vertices that are not consecutive on the cycle. For example, the hexagon shown in Figure 9 is a cycle without a chord. In an interval graph, it should not be allowed. In fact, it is forbidden.

Let's see why it should be forbidden. If I were to try to construct an interval representation for the cycle, what would happen? I would have to start somewhere by drawing an interval, and then I would have to draw the interval of its neighbor, which intersects it, and then continue to its neighbor, which intersects the second one but not the first one, and so forth, as illustrated in Figure 9. The fourth has to be disjoint from

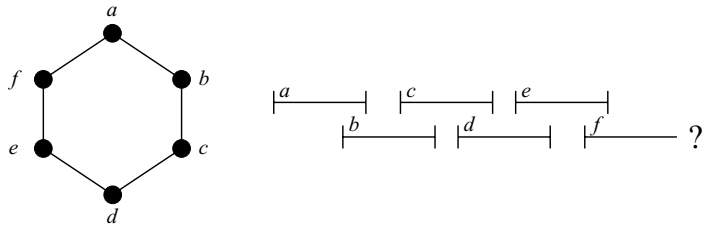


Figure 9. A cycle without a chord.

the second but hit the third, and the fifth has to hit the fourth but not the third. Finally, the sixth has to hit the fifth and not the fourth, yet somehow must close the loop and intersect the first. This cannot be done because we must draw intervals on a line. Thus, it is impossible to get an interval representation for this or any chordless cycle. It is a forbidden configuration. *A chordless cycle cannot be part of an interval graph.*

The second property of interval graphs is the transitive orientation property of the complement or co-TRO. Recall that the edges of the complement of an interval graph represent disjoint intervals. Since in a pair of disjoint intervals, one appears totally before the other, we may orient the associated edge in the disjointness graph from the later to the earlier. It is easy to verify that such an orientation is transitive: if a is before b, and b is before c, then a is before c. Now here is the punch line, a characterization theorem of Gilmore and Hoffman [10] from 1964.

Theorem 1 *A graph G is an interval graph if and only if G is chordal and its complement* $\overline{G}$ *is transitively orientable.*

Additional characterizations of interval graphs can be found in the books [1, 2, 3]. Next, we will illustrate the use of some of these properties to reason about time intervals in solving the Berge Mystery Story.

5. The Berge Mystery Story

Some of you who have read my first book, *Algorithmic Graph Theory and Perfect Graphs*, know the Berge mystery story. For those who don't, here it is:

> Six professors had been to the library on the day that the rare tractate was stolen. Each had entered once, stayed for some time and then left. If two were in the library at the same time, then at least one of them saw the other. Detectives questioned the professors and gathered the following testimony: Abe said that he saw Burt and Eddie in the library; Burt said that he saw Abe and Ida; Charlotte claimed to have seen Desmond and Ida; Desmond said that he saw Abe and Ida; Eddie testified to seeing Burt and Charlotte; Ida said that she saw Charlotte and Eddie. One of the professors lied!! Who was it?

Let's pause for a moment while you try to solve the mystery. Being the interrogator, you begin, by collecting the data from the testimony written in the story: Abe saw Burt and Eddie, Burt saw Abe and Ida, etc. Figure 10(a) shows this data with an arrow pointing from X to Y if X "claims" to have seen Y. Graph theorists will surely start attacking this using graph theory. How can we use it to solve the mystery?

Remember that the story said each professor came into the library, was there for an interval of time, during that interval of time he saw some other people. If he saw somebody, that means their intervals intersected. So that provides some data about the intersection, and we can construct an intersection graph G, as in Figure 10(b). This graph "should be" an interval graph if all the testimony was truthful and complete.

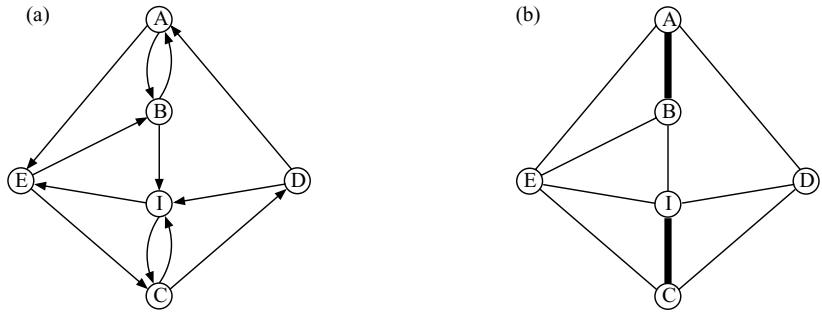


Figure 10. The testimony graphs.

However, we know that there is a lie here. Why? Because looking at the intersection graph G, we see a chordless cycle of length four which is an impossibility. This is supposed to be an interval graph, so we know something is wrong.

Notice in Figure 10(a), that some pairs have arrows going in both directions, for example, Burt saw Abe and Abe saw Burt, and other pairs are just one way. That gives us some further information. Some of the edges in the intersection graph are more confident edges than others. A bold black edge in Figure 10(b) indicates a double arrow in Figure 10(a), and it is pretty confident because B saw A and A saw B, so, if at least one of them is telling the truth, the edge really exists. Similarly, for I and C. But all the one-way arrows are possibly true and possibly false. How shall we argue? Well, if we have a 4-cycle, one of those four professors is the liar. I do not know which one, so I will list all the cycles and see who is common. ABID is a cycle of length 4 without a chord; so is ADIE. There is one more – AECD – that is also a 4-cycle, with no chord. What can we deduce? We can deduce that the liar is one of these on a 4-cycle. That tells us Burt is not a liar. Why? Burt is one of my candidates in the first cycle, but he is not a candidate in the second, so he is telling the truth. The same goes for Ida; she is not down in the third cycle, so she is also telling the truth. Charlotte is not in the first cycle, so she is ok. The same for Eddie, so he is ok. Four out of the six professors are now known to be telling the truth. Now it is only down to Abe and Desmond. What were to happen if Abe is the liar? If Abe is the liar, then ABID still remains a cycle because of the testimony of Burt, who is truthful. That is, suppose Abe is the liar, then Burt, Ida and Desmond would be truth tellers and ABID would still be a chordless cycle, which is a contradiction. Therefore, Abe is not the liar. The only professor left is Desmond. Desmond is the liar.

Was Desmond Stupid or Just Ignorant?

If Desmond had studied algorithmic graph theory, he would have known that his testimony to the police would not hold up. He could have said that he saw everyone, in which case, no matter what the truthful professors said, the graph would be an interval graph. His (false) interval would have simply spanned the whole day, and all the data would be consistent. Of course, the detectives would probably still not believe him.

6. Many Other Families of Intersection Graphs

We have seen a number of applications of interval graphs, and we will see one more a little bit later. However, I now want to talk about other kinds of intersections – not just of intervals – to give an indication of the breadth of research that goes on in this area. There is a mathematician named Victor Klee, who happened to have been Robert Jameson's thesis advisor. In a paper in the American Mathematics Monthly in 1969, Klee wrote a little article that was titled "What are the intersection graphs of arcs in a circle?" [21]. At that point in time, we already had Gilmore and Hoffman's theorem characterizing interval graphs and several other theorems of Lekkerkerker and Boland, Fulkerson and Gross, Ghouila-Houri and Berge (see [1]). Klee came along and said, *Okay, you've got intervals on a line, what about arcs going along a circle?* Figure 11 shows a model of arcs on a circle, together with its intersection graph. It is built in a similar way as an interval graph, except that here you have arcs of a circle instead of intervals of a line. Unlike interval graphs, circular arc intersection graphs may have chordless cycles. Klee wanted to know: Can you find a mathematical characterization for these circular arc graphs?

In fact, I believe that Klee's paper was really an implicit challenge to consider a whole variety of problems on many kinds of intersection graphs. Since then, dozens of researchers have begun investigating intersection graphs of boxes in the plane, paths in a tree, chords of a circle, spheres in 3-space, trapezoids, parallelograms, curves of functions, and many other geometrical and topological bodies. They try to recognize them, color them, find maximum cliques and independent sets in them. (I once heard someone I know mention in a lecture, "A person could make a whole career on algorithms for intersection graphs!" Then I realized, that person was probably me.)

Circular arc graphs have become another important family of graphs. Renu Laskar and I worked on domination problems and circular arc graphs during my second visit to Clemson in 1989, and published a joint paper [14].

On my first visit to Clemson, which was in 1981, I started talking to Robert Jamison about an application that comes up on a tree network, which I will now describe.

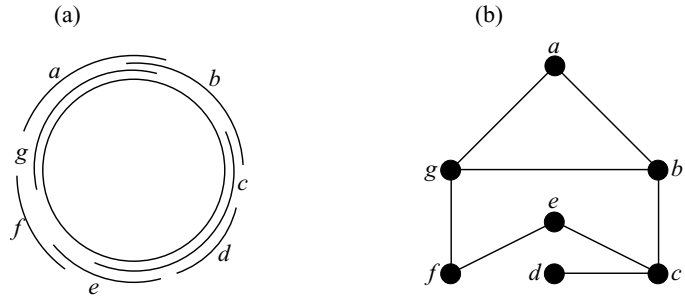


Figure 11. (a) Circular arc representation. (b) Circular arc graph.

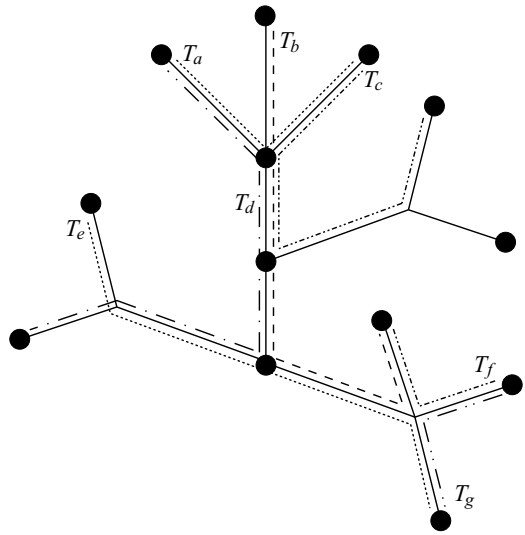


Figure 12. A representation of paths in a tree.

Figure 12 shows a picture of a tree, the black tree, which you may want to think of as a communication network connecting different places. We have pairs of points on the tree that have to be connected with paths – green paths, red paths and purple paths. They must satisfy the property that if two of these paths overlap, even a little bit, this intersection implies that they conflict and we cannot assign the same resource to them at the same time. I am interested in the intersection graph of these paths. Figure 13 gives names to these paths and shows their intersection graph. As before, if two paths intersect, you connect the two numbers by an edge, and if they are disjoint, you do not. Coloring this graph is the same as assigning different colors to these paths; if they intersect, they get different colors. I can interpret each color to mean a time slot when the path has exclusive use of the network. This way there is the red time slot, the purple

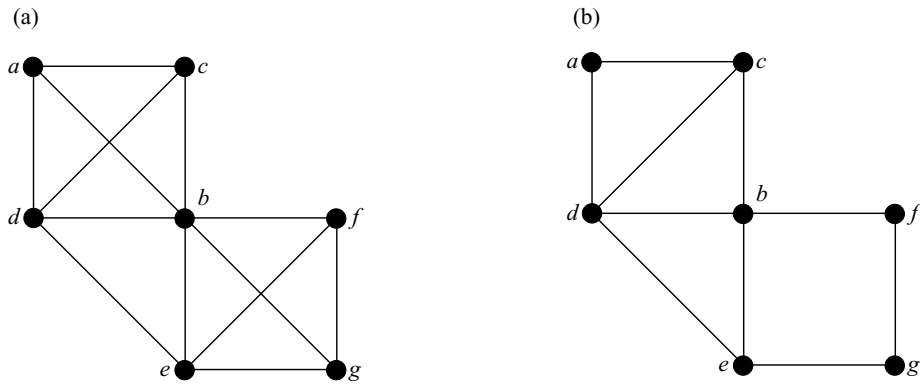


Figure 13. (a) Vertex Intersection Graph (VPT) and (b) Edge Intesection Graph (EPT), both of the paths shown on Figure 12.

Table 1. Six graph problems and their complexity on VPT graphs and EPT graphs.

Graph Problem	VPT graphs	EPT graphs
recognition	polynomial	NP-complete [12]
maximum independent set	polynomial	polynomial [25]
maximum clique	polynomial	polynomial [11]
minimum coloring	polynomial	NP-complete [11]
3/2 approximation coloring	polynomial	polynomial [25]
minimum clique cover	polynomial	NP-complete [11]

time slot, etc. All the red guys can use the network at the same time; all the purple guys can use it together some time later; brown guys use it at yet a different time. We might be interested to find (*i*) a maximum independent set, which would be the largest number of paths to be used simultaneously, or (*ii*) a minimum coloring, which would be a schedule of time periods for all of the paths.

I began investigating the intersection graphs of paths and trees, and immediately had to look at two kinds of intersections – one was sharing a vertex and one was sharing an edge. This gave rise to two classes of graphs, which we call vertex intersection graphs of paths of a tree (VPT graphs) and edge intersection graphs of paths of a tree (EPT graphs), quickly observing that they are different classes – VPT graphs are chordal and perfect, the EPT graphs are not.

After discussing this at Clemson, Robert and I began working together on EPT graphs, a collaboration of several years resulting in our first two joint papers [11, 12]. We showed a number of mathematical results for EPT graphs, and proved several computational complexity results. Looking at the algorithmic problems – recognition, maximum independent set, maximum clique, minimum coloring, 3/2 approximation (Shannon) coloring, and minimum clique cover – all six problems are polynomial for vertex intersection (VPT graphs), but have a real mixture of complexities for edge intersection (EPT graphs), see Table 1. More recent extensions of EPT graphs have been presented in [15, 20].

There are still other intersection problems you could look at on trees. Here is an interesting theorem that some may know. If we consider an intersection graph of subtrees of a tree, not just paths but arbitrary subtrees, there is a well known characterization attributed to Buneman, Gavril, and Wallace discovered independently by each of them in the early 1970's, see [1].

Theorem 2 *A graph G is the vertex intersection graph of subtrees of a tree if and only if it is a chordal graph.*

Here is another Clemson connection. If you were to look at subtrees not of just any old tree, but of a special tree, namely, a star, you would get the following theorem of Fred McMorris and Doug Shier [23] from 1983.

Table 2. Graph classes involving trees.

Type of Interaction	Objects	Host	Graph Class
vertex intersection	subtrees	tree	chordal graphs
vertex intersection	subtrees	star	split graphs
edge intersection	subtrees	star	all graphs
vertex intersection	paths	path	interval graphs
vertex intersection	paths	tree	path graphs or VPT graphs
edge intersection	paths	tree	EPT graphs
containment	intervals	line	permutation graphs
containment	paths	tree	? (open question)
containment	subtrees	star	comparability graphs

Theorem 3 *A graph G is a vertex intersection graph of distinct subtrees of a star if and only if both G and its complement $\overline{G}$ are chordal.*

Notice how well the two theorems go together: If the host tree is any tree, you get chordal, and if it is a star, you get chordal ∩ co-chordal, which are also known as split graphs. In the case of edge intersection, the chordal graphs are again precisely the edge intersection graphs of subtrees of a tree, however, *every possible graph can be represented as the edge intersection graph of subtrees of a star*. Table 2 summarizes various intersection families on trees. Some of them may be recognizable to you; for those that are not, a full treatment can be found in Chapter 11 of [4].

7. Tolerance Graphs

The grandfather of all intersection graph families is the family of interval graphs. Where do we go next? One direction has been to measure the size of the intersection and define a new class called the interval tolerance graphs, first introduced by Golumbic and Monma [16] in 1982. It is also the topic of the new book [4] by Golumbic and Trenk. We also go into trapezoid graphs and other kinds of intersection graphs.

Even though I am not going to be discussing tolerance graphs in detail, I will briefly state what they are and in what directions of research they have taken us. In particular, there is one related class of graphs (NeST) that I will mention since it, too, has a Clemson connection.

An undirected graph $G = (V, E)$ is a *tolerance graph* if there exists a collection $\mathcal{I} = \{I_v\}_{v \in V}$ of closed intervals on the real line and an assignment of positive numbers $t = \{t_v\}_{v \in V}$ such that

$$vw \in E \Leftrightarrow |I_v \cap I_w| \geq \min\{t_v, t_w\}.$$

Here $|I_u|$ denotes the length of the interval I_u. The positive number t_v is called the *tolerance* of v, and the pair $\langle \mathcal{I}, t \rangle$ is called an *interval tolerance representation* of G. Notice that interval graphs are just a special case of tolerance graphs, where

each tolerance t_v equals some sufficiently small $\epsilon > 0$. A tolerance graph is said to be *bounded* if it has a tolerance representation in which $t_v \leq |I_v|$ for all $v \in V$.

The definition of tolerance graphs was motivated by the idea that a small, or "tolerable" amount of overlap, between two intervals may be ignored, and hence not produce an edge. Since a tolerance is associated to each interval, we put an edge between a pair of vertices when at least one of them (the one with the smaller tolerance) is "bothered" by the size of the intersection.

Let's look again at the scheduling problem in Figure 5. In that example, the chief university officer of classroom scheduling needs four rooms to assign to the six lectures. But what would happen if she had only three rooms available? In that case, would one of the lectures c, d, e or f have to be cancelled? Probably so. However, suppose some of the professors were a bit more tolerant, then an assignment might be possible.

Consider, in our example, if the tolerances (in minutes) were:

$$t_a = 10, t_b = 5, t_c = 65, t_d = 10, t_e = 20, t_f = 60.$$

Then according to the definition, lectures c and f would no longer conflict, since $|I_c \cap I_f| \leq 60 = \min\{t_c, t_f\}$. Notice, however, that lectures e and f remain in conflict, since Professor e is too intolerant to ignore the intersection. The tolerance graph for these values would therefore only erase the edge cf in Figure 6, but this is enough to admit a 3-coloring.

Tolerance graphs generalize both interval graphs and another family known as permutation graphs. Golumbic and Monma [16] proved in 1982 that every bounded tolerance graph is a cocomparability graph, and Golumbic, Monma and Trotter [17] later showed in 1984 that tolerance graphs are perfect and are contained in the class of weakly chordal graphs. Coloring bounded tolerance graphs in polynomial time is an immediate consequence of their being cocomparability graphs. Narasimhan and Manber [24] used this fact in 1992 (as a subroutine) to find the chromatic number of any (unbounded) tolerance graph in polynomial time, but not the coloring itself. Then, in 2002, Golumbic and Siani [19] gave an $O(qn + n \log n)$ algorithm for coloring a tolerance graph, given the tolerance representation with q vertices having unbounded tolerance. For details and all the references, see Golumbic and Trenk [4]. The complexity of recognizing tolerance graphs and bounded tolerance graphs remain open questions.

A several "variations on the theme of tolerance" in graphs have been defined and studied over the past years. By substituting a different "host" set instead of the real line, and then specifying the type of subsets of that host to consider instead of intervals, along with a way to measure the size of the intersection of two subsets, we obtain other classes of tolerance-type graphs, such as neighborhood subtree tolerance (NeST) graphs (see Section 8 below), tolerance graphs of paths on a tree or tolerance competition graphs. By changing the function *min* for a different binary function ϕ (for example, *max, sum, product*, etc.), we obtain a class that will be called ϕ*-tolerance* graphs. By replacing

the measure of the length of an interval by some other measure μ of the intersection of the two subsets (for example, cardinality in the case of discrete sets, or number of branching nodes or maximum degree in the case of subtrees of trees), we could obtain yet other variations of tolerance graphs. When we restrict the tolerances to be 1 or ∞, we obtain the class of *interval probe graphs*. By allowing a separate leftside tolerance and rightside tolerance for each interval, various bitolerance graph models can be obtained. For example, Langley [22] in 1993 showed that *the bounded bitolerance graphs are equivalent to the class of trapezoid graphs*. Directed graph analogues to several of these models have also been defined and studied. For further study of tolerance graphs and related topics, we refer the reader to Golumbic and Trenk [4].

8. Neighborhood Subtree Tolerance (NeST) Graphs

On my third visit to Clemson, also in 1989, Lin Dearing told me about a class of graphs that generalized tolerance graphs, called neighborhood subtree tolerance (NeST) graphs. This generalization consists of representing each vertex $v \in V(G)$ of a graph G by a subtree T_v of a (host) tree embedded in the plane, where each subtree T_v has a center c_v and a radius r_v and consists of all points of the host tree that are within a distance of r_v from c_v. The size of a neighborhood subtree is twice its radius, or its diameter. The size of the intersection of two subtrees T_u and T_v is the Euclidean length of a longest path in the intersection, namely, the diameter of the subtree $T_u \cap T_v$. Bibelnieks and Dearing [9] investigated various properties of NeST graphs. They proved that bounded NeST graphs are equivalent to proper NeST graphs, and a number of other results. You can see their result in one of the boxes in Figure 14.

I bring this to your attention because it is typical of the research done on the relationships between graph classes. We have all these classes, some of which are arranged in a containment hierarchy and others are equivalent, shown in the same box of the figure. Figure 14 is an example of an incomplete hierarchy since some of the relationships are unknown. Interval graphs and trees are the low families on this diagram. They are contained in the classes above them, which are in turn contained in the ones above them, etc. The fact that NeST is contained in weakly chordal graphs is another Clemson result from [9].

You see in Figure 14 a number of question marks. Those are the open questions still to be answered. So as long as the relationships between several graph classes in the hierarchy are not known yet (this is page 222 of [4]), we remain challenged as researchers.

9. Interval Probe Graphs

Lastly, I want to tell you about another class of graphs called the interval probe graphs. They came about from studying interval graphs, where some of the adjacency information was missing. This is a topic that is of recent interest, motivated by computational biology applications. The definition of an interval probe graph is a graph

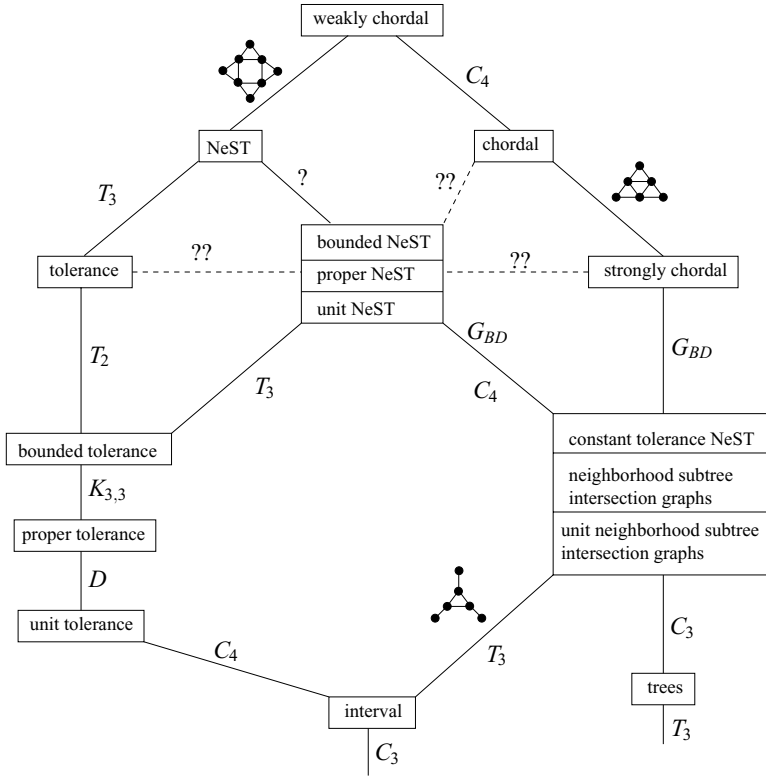


Figure 14. The NeST Hierarchy (reprinted from [4]).

whose vertices are partitioned into two sets: the probes P and non-probes N, where N is an independent set, and there must exist a(n interval) completion, by adding some extra edges between nodes in N so that this augmented graph is an interval graph. The names probe and non-probe come from the biological application. Partitioned into these two sets, the edges between pairs of P nodes and between P and N nodes are totally known, but there is nothing known about the connections between pairs of N nodes. Is it possible to fill in some of these missing edges in order to get an interval representation?

That is the mathematical formulation of it. You can ask, "What kinds of graphs do you have in this class?" Figure 15(a) shows an example of an interval probe graph and a representation for it; The black vertices are probes and the white vertices are non–probes. Figure 15(b) gives an example of a graph that is not interval probe, no matter how the vertices may be partitioned. I will let you prove this on your own, but if you get stuck, then you can find many examples and proofs in Chapter 4 of the Tolerance Graphs book [4].

I will tell you a little bit about how this problem comes about in the study of genetic DNA sequences. Biologists want to be able to know the whole structure of the DNA

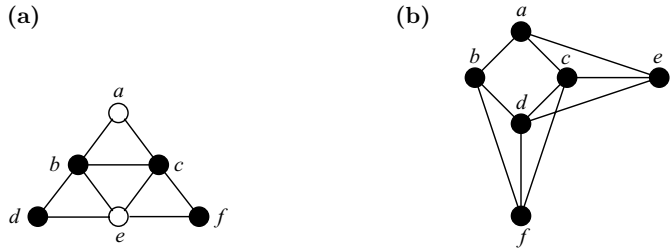


Figure 15. An example of an interval probe graph and a non-interval probe graph.

of some gene, which can be regarded as a long string of about a million letters. The Human Genome Project was to be able to identify the sequence of those strings in people. The biologists worked together with other scientists and mathematicians and computer scientists and so on, and here is one method that they use. They take the gene and they place it in a beaker with an enzyme, and then dump it out on the floor where it breaks into a bunch of pieces. These fragments are called gene fragments. Now they will take the same gene and put in a different enzyme, stir it and shake it and then dump it on the floor where it breaks up in a different way. They do it again and again with a bunch of different enzymes.

Now we can think of the problem as reconstructing several puzzles, each one having different pieces, but giving the same completed string. One could argue, "Ah, I have a string that says ABBABBABBA and someone else has a similar string BABBACCADDA and they would actually overlap nicely." By cooperating maybe we could put the puzzle back together, by recombining overlapping fragments to find the correct ordering, that is, if we are able to do all this pattern matching.

Imagine, instead, there is some experiment where you take this little piece of fragment and you can actually test somehow magically, (however a biologist tests magically), how it intersects with the other fragments. This gives you some intersection data. Those that are tested will be the probes. In the interval probe model, for every probe fragment we test, we know exactly whom he intersects, and for the unlucky fragments that we do not test, we know nothing regarding the overlap information between them. They are the non-probes. This is the intersection data that we get from the biologist. Now the interval probe question is: can we somehow fill-in the missing data between the pairs of non-probes so that we can get a representation consistent with that data?

Here is a slightly different version of the same problem – played as a recognition game. Doug has an interval graph H whose edges are a secret known only to him. A volunteer from the audience chooses a subset N of vertices, and Doug draws you a graph G by secretly erasing from H all the edges between pairs of vertices in N, making N into an independent set. My game #1 is, if we give you the graph G and the independent set N, can you fill-in some edges between pairs from N and rebuild an interval graph (not necessarily H)?

This problem can be shown to be solvable in time proportional to n^2 in a method that was found by Julie Johnson and Jerry Spinrad, published in SODA 2001. The following year there was a faster algorithm by Ross McConnell and Jerry Spinrad that solved the problem in time $O(m \log n)$, published in SODA 2002. Here n and m are the number of vertices and edges, respectively.

There is a second version of the game, which I call the unpartitioned version: this time we give you G, but we do not tell you which vertices are in N. My game #2 requires both choosing an appropriate independent set and filling in edges to complete it to an interval graph. So far, the complexity of this problem is still an open question. That would be recognizing unpartitioned interval probe graphs.

10. The Interval Graph Sandwich Problem

Interval problems with missing edges, in fact, are much closer to the problem Seymour Benzer originally addressed. He asked the question of reconstructing an interval model even when the probe data was only partially known. Back then, they could not answer his question, so instead he asked the 'simpler' interval graph question: "Suppose I had *all* of the intersection data, then can you test consistency and give me an interval representation?" It was not until much later, in 1993, that Ron Shamir and I gave an answer to the computational complexity of Benzer's real question.

You are given a partially specified graph, i.e., among all possible pairs of vertices, some of the pairs are definitely edges, some of them are definitely non-edges, and the remaining are unknown. Can you fill-in some of the unknowns, so that the result will be an interval graph? This problem we call the *interval sandwich problem* and it is a computationally hard problem, being NP-complete [18].

For further reading on sandwich problems, see [13], Chapter 4 of [4] and its references.

11. Conclusion

The goal of this talk has been to give you a feeling for the area of algorithmic graph theory, how it is relevant to applied mathematics and computer science, what applications it can solve, and why people do research in this area.

In the world of mathematics, sometimes I feel like a dweller, a permanent resident; at other times as a visitor or a tourist. As a mathematical resident, I am familiar with my surroundings. I do not get lost in proofs. I know how to get around. Yet, sometimes as a dweller, you can become jaded, lose track of what things are important as things become too routine. This is why I like different applications that stimulate different kinds of problems. The mathematical tourist, on the other hand, may get lost and may not know the formal language, but for him everything is new and exciting and interesting. I hope

that my lecture today has given both the mathematical resident and the mathematical tourist some insight into the excitement and enjoyment of doing applied research in graph theory and algorithms.

12. Acknowledgements

I am honored to have been invited to deliver the 2003 Andrew F. Sobczyk Memorial Lecture. This has been my sixth visit to Clemson University, and I would like to thank my hosts from the Mathematics Department for their generous hospitality. Clemson University is a leader in the field of discrete mathematics, and building and maintaining such leadership is no small achievement. It takes hard work and a lot of excellent research. I salute those who have made this possible, and who have influenced my career as well: Renu Laskar, Robert Jamison, P.M. (Lin) Dearing, Doug Shier, Joel Brawley and Steve Hedetniemi. I would also like to thank Sara Kaufman and Guy Wolfovitz from the Univerity of Haifa for assisting with the written version of this lecture.

Further Reading

[1] M.C. Golumbic, "*Algorithmic Graph Theory and Perfect Graphs*", Academic Press, New York, 1980. Second edition: *Annals of Discrete Mathematics* 57, Elsevier, Amsterdam, 2004.
This has now become the classic introduction to the field. It conveys the message that intersection graph models are a necessary and important tool for solving real-world problems for a large variety of application areas, and on the mathematical side, it has provided rich soil for deep theoretical results in graph theory. In short, it remains a stepping stone from which the reader may embark on one of many fascinating research trails. The second edition of *Algorithmic Graph Theory and Perfect Graphs* includes a new chapter called *Epilogue 2004* which surveys much of the new research directions from the Second Generation. Its intention is to whet the appetite.
Seven other books stand out as the most important works covering advanced research in this area. They are the following, and are a must for any graph theory library.

[2] A. Brandstädt, V.B. Le and J.P. Spinrad, "*Graph Classes: A Survey*", SIAM, Philadelphia, 1999,
This is an extensive and invaluable compendium of the current status of complexity and mathematical results on hundreds of families of graphs. It is comprehensive with respect to definitions and theorems, citing over 1100 references.

[3] P.C. Fishburn, "*Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*", John Wiley & Sons, New York, 1985.
Gives a comprehensive look at the research on this class of ordered sets.

[4] M.C. Golumbic and A.N. Trenk, "*Tolerance Graphs*", Cambridge University Press, 2004.
This is the youngest addition to the perfect graph bookshelf. It contains the first thorough study of tolerance graphs and tolerance orders, and includes proofs of the major results which have not appeared before in books.

[5] N.V.R. Mahadev and U.N. Peled, "*Threshold Graphs and Related Topics*", North-Holland, 1995.
A thorough and extensive treatment of all research done in the past years on threshold graphs (Chapter 10 of Golumbic [1]), threshold dimension and orders, and a dozen new concepts which have emerged.

[6] T.A. McKee and F.R. McMorris, "*Topics in Intersection Graph Theory*", SIAM, Philadelphia, 1999.
A focused monograph on structural properties, presenting definitions, major theorems with proofs and many applications.

[7] F.S. Roberts, "*Discrete Mathematical Models, with Applications to Social, Biological, and Environmental Problems*", Prentice-Hall, Engelwood Cliffs, New Jersey, 1976.
This is the classic book on many applications of intersection graphs and other discrete models.

[8] W.T. Trotter, "*Combinatorics and Partially Ordered Sets*", Johns Hopkins, Baltimore, 1992.
Covers new directions of investigation and goes far beyond just dimension problems on ordered sets.

Additional References

[9] E. Bibelnieks and P.M. Dearing, Neighborhood subtree tolerance graphs, *Discrete Applied Math.* 43: 13–26 (1993).

[10] P.C. Gilmore and A.J. Hoffman, A characterization of comparability graphs and of interval graphs. *Canad. J. Math.* 16: 539–548 (1964).

[11] M.C. Golumbic and R.E. Jamison, The edge intersection graphs of paths in a tree, *J. Combinatorial Theory, Series B* 38: 8–22 (1985).

[12] M.C. Golumbic and R.E. Jamison, Edge and vertex intersection of paths in a tree, *Discrete Math.* 55: 151–159 (1985).

[13] M.C. Golumbic and H. Kaplan and R. Shamir, Graph sandwich problems, *J. of Algorithms* 19: 449–473 (1995).

[14] M.C. Golumbic and R.C. Laskar, Irredundancy in circular arc graphs, *Discrete Applied Math.* 44: 79–89 (1993).

[15] M.C. Golumbic, M. Lipshteyn and M. Stern, The k-edge intersection graphs of paths in a tree, *Congressus Numerantium* (2004), to appear.

[16] M.C. Golumbic and C.L. Monma, A generalization of interval graphs with tolerances", *Congressus Numerantium* 35: 321–331 (1982).

[17] M.C. Golumbic and C.L. Monma and W.T. Trotter, Tolerance graphs, *Discrete Applied Math.* 9: 157–170 (1984).

[18] M.C. Golumbic and R. Shamir, Complexity and algorithms for reasoning about time: A graph theoretic approach, *J. Assoc. Comput. Mach.* 40: 1108–1133 (1993).

[19] M.C. Golumbic and A. Siani, Coloring algorithms and tolerance graphs: reasoning and scheduling with interval constraints, *Lecture Notes in Computer Science* 2385, Springer-Verlag, (2002) pp. 196–207.

[20] R.E. Jamison and H.M. Mulder, Constant tolerance representations of graphs in trees, *Congressus Numerantium* 143: 175–192 (2000).

[21] V. Klee, What are the intersection graphs of arcs in a circle? *American Math. Monthly* 76: 810–813 (1969).

[22] L. Langley, *Interval tolerance orders and dimension*, Ph.D. Thesis, Dartmouth College", June, (1993).

[23] F.R. McMorris and D.R. Shier, Representing chordal graphs on $K_{1,n}$, *Comment. Math. Univ. Carolin.* 24: 489–494 (1983).

[24] G. Narasimhan and R. Manber, Stability number and chromatic number of tolerance graphs, *Discrete Applied Math.* 36: 47–56 (1992).

[25] R.E. Tarjan, Decomposition by clique separators, *Discrete Math.* 55: 221–232 (1985).

4

Decompositions and Forcing Relations in Graphs and Other Combinatorial Structures

Ross M. McConnell
Department of Computer Science
Colorado State University
Fort Collins, CO 80523-1873 USA

1. Introduction

A textbook introduction to many of the ideas that are the focus of this chapter is given in the now-classic textbook of Golumbic [9]. The main focus of the chapter is to show that many of these problems can be brought under a single abstraction, which involves decompositions satisfying a set of axioms on a variety of combinatorial structures, and their dual representations with a forcing relation on elements of the structures. One of these is the *modular decomposition*, where the combinatorial structure is an undirected graph, and its well-known dual, the *Gamma forcing relation*, on the edges of the graph. This was described by Gallai [8] and is dealt with extensively in Golumbic's book.

The abstraction for the decompositions were put forth by Möhring [29], who recognized their reappearance in a variety of domains, such as hypergraphs and boolean functions. The ones I describe here are newer examples, and what is particularly surprising is that, unlike Möhring's examples, they are also accompanied by an analog of Gallai's Gamma forcing relation.

The exact details of the decomposition and its dual forcing relation depend on the combinatorial structure under consideration. However, the techniques that are used to solve a problem in one of the contexts are often driven by the axioms satisfied by the decomposition and its dual forcing relation in the other contexts where the abstraction applies. In particular, I have been able to adapt techniques developed by Jerry Spinrad and me [21] to obtain linear time bounds for modular decomposition and its dual *transitive orientation* to obtain linear time bounds for problems on other structures where I have found new examples of the abstraction. These include linear time bounds

for recognition of *circular-arc graphs* [23], near-linear time bounds, together with Spinrad, for recognition of *probe interval graphs* [25], so-called *certifying algorithms* for recognizing *permutation graphs* and matrices with the *consecutive-ones property*, as well as algorithms for a variety of combinatorial problems that are beyond the scope of this chapter, such as modular decomposition of directed graphs with Fabien de Montgolfier [24], and a variety of problems on collections of linear orders, also with de Montgolfier, currently under review, and a representation of all *circular-ones orderings* of a 0-1 matrix, with Wen-Lian Hsu [12]

The **intersection graph** of a family of n sets is the graph where the vertices are the sets, and the edges are the pairs of sets that intersect. Every graph is the intersection graph of some family of sets [16]. A graph is an **interval graph** if there is a way to order the universe from which the sets are drawn so that each set is consecutive in the ordering. Equivalently, a graph is an interval graph if it is the intersection graph of a finite set of intervals on a line.

A graph is a **circular-arc graph** if it is the intersection graph of a finite set of arcs on a circle. (See Figure 1.) A **realizer** of an interval graph or circular-arc graph G is a set of intervals or circular arcs that represent G in this way.

An interval graph is a special case of circular-arc graphs; it is a circular-arc graph that can be represented with arcs that do not cover the entire circle. Some circular-arc graphs do not have such a representation, so the class of interval graphs is a proper subclass of the class of circular-arc graphs.

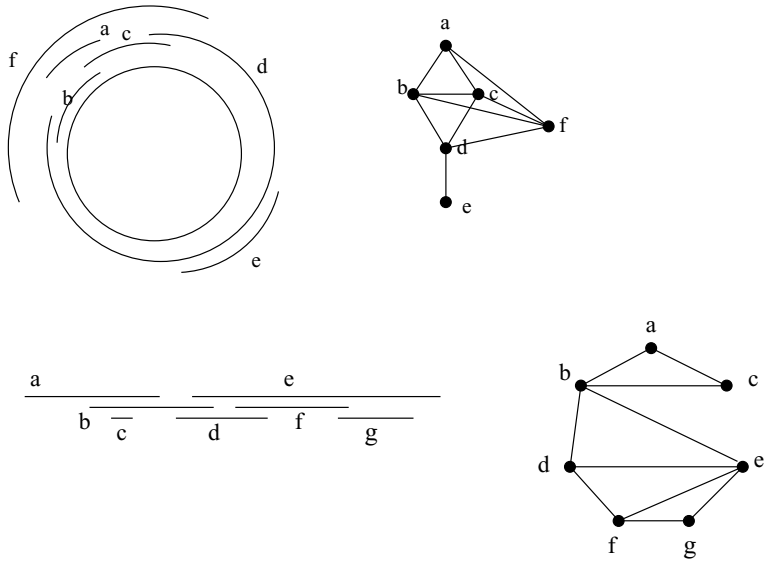


Figure 1. A circular-arc graph is the intersection graph of a set of arcs on the circle, while an interval graph is the intersection graph of a set of intervals on the line.

Interval graphs and circular-arc graphs arise in scheduling problems and other combinatorial problems. Before the structure of DNA was well-understood, Seymour Benzer [1] was able to show that the set of intersections of a large number of fragments of genetic material in a virus were an interval graph. This provided strong evidence that genetic information was arranged inside a structure with a linear topology.

Being able to determine whether a graph is an interval graph or circular-arc graph constitutes **recognition** of these graph classes. However, having a representation of a graph with intervals or arcs can be helpful in solving combinatorial problems on the graph, such as isomorphism testing and finding maximum independent sets and cliques [4, 11]. Therefore, a stronger result than just recognizing the class is being able to produce the representation whenever a graph is a member of the class. In addition to its other uses, the representation constitutes a certificate that the graph is a member of the class, which allows the output of a distrusted implementation of the recognition algorithm to be checked.

In the 1960's, Fulkerson and Gross [7] gave an $O(n^4)$ algorithm for finding an interval representation of an interval graph. Booth and Lueker improved this to linear-time in the 1970's [3]. At that time, Booth conjectured that recognition of circular-arc graphs would turn out to be NP-complete [2]. Tucker disproved this with an $O(n^3)$ algorithm [34]. However, a linear-time bound has only been given recently [17, 23].

A directed graph is **transitive** if, whenever (x, y) and (y, z) are directed edges, (x, z) is also a directed edge. Transitive graphs are a natural way to represent a **partial order**, or **poset** relation, which is a relation that is reflexive, antisymmetric, and transitive. An example of a poset relation is the subset relation on a family of sets: if X, Y, and Z are members of the family, then $X \subseteq X$, hence, it is reflexive; if $X \subseteq Y$ and $Y \subseteq Z$, then $X \subseteq Z$, hence it is transitive; and if $X \neq Y$, at most one of $X \subseteq Y$ and $Y \subseteq X$ can apply, hence it is antisymmetric.

A poset relation R has an associated symmetric **comparability relation** $\{(X, Y) | XRY \text{ or } YRX\}$. For instance, the comparability relation associated with the subset relation on a set family is the relation $\{(X, Y) | X$ and Y are members of the family and one of X and Y contains the other$\}$. A comparability relation can be modeled in an obvious way with an undirected graph. Such a graph is a **comparability graph**.

Given a poset relation in the form of a transitive digraph, it is trivial to get the associated comparability graph: one just makes each edge symmetric. Given a comparability graph, it is not so easy to find an associated poset relation. This problem can be solved by assigning an orientation to the edges of the comparability graph so that the resulting digraph is transitive (see Figure 2). Work on algorithms for this problem began in the early 1960's, but a linear-time time algorithm was not given until the the late 1990's. [20, 21].

Somewhat surprisingly, this time bound for transitive orientation does not give an algorithm for recognizing comparability graphs. On an input graph that is not a

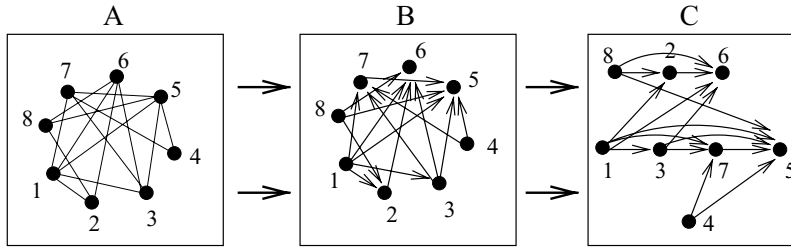


Figure 2. A transitive orientation of a graph (A) is an orientation of the edges so that the resulting directed graph is transitive (B). Figure (C) is a redrawing of (B) that makes it easy to see that the orientation is transitive. A **comparability graph** is any graph that has a transitive orientation.

comparability graph, the algorithm produces an orientation that is not transitive. Currently, there is no known linear-time algorithm for recognizing whether a digraph is transitive. This result is nevertheless strong enough to solve a number of other long-standing problems in linear time in a stricter sense, that is, so that one can recognize whether the output is correct even when one is not certain whether the input meets the required preconditions.

An example of such a problem is that of finding a maximum clique and **minimum proper vertex coloring** of a comparability graph. A proper vertex coloring is an assignment of labels to the vertices so that no adjacent pairs have the same color, and it is minimum if it minimizes the number of colors used. The size of a clique is a lower bound on the number of colors needed, since the vertices of a clique must all be different colors.

These problems are NP-hard on graphs in general, but polynomially solvable on comparability graphs. If G is a comparability graph, a longest directed path in the transitive orientation must be a maximum clique. Labeling each vertex with the length of the longest directed path originating at the vertex gives a proper coloring of the same size, which serves to prove that the clique is a maximum one and the vertex coloring is a minimum one (see Figure 3).

Another example of such a problem is that of finding an interval realizer of an interval graph. It is easy to see that if I is a set of intervals on the real line, then the relation $P_i = \{(X, Y) | X$ and Y are disjoint elements of I and X precedes $Y\}$ is a poset relation. It follows from this observation that the complement of an interval graph is a comparability graph. One way to produce an interval representation of an interval graph is to find a transitive orientation of the complement, using linear extensions to represent them, in order to avoid exceeding linear storage. This gives P_I for some interval representation of the graph, and it is not hard to see how to construct an interval representation from it [21].

This process always succeeds if G is an interval graph. If G is not known to be an interval graph, one cannot tell in linear time whether the transitive orientation

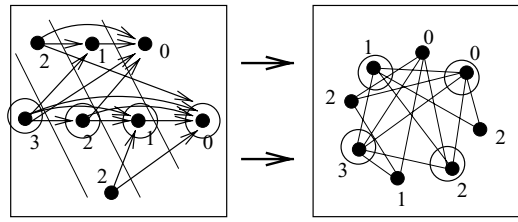


Figure 3. To find a maximum clique and a minimum coloring of a comparability graph, find a transitive orientation, and then label each vertex with the length of the longest directed path originating at the vertex. The labeling of the vertices is a proper coloring, and, because the orientation is transitive, every directed path is a clique. A longest directed path is therefore a clique whose size is equal to the number of colors used. The size of a clique is a lower bound on the number of colors required to color a graph, so the coloring proves that the clique is a maximum one, and the clique proves that the coloring is a minimum one.

algorithm produced a transitive orientation of the complement of G. Nevertheless, there is no interval representation of G, so the construction of the interval representation must fail, and this can be detected in linear time.

An ordering $(x_1, x_2, \ldots, x_n)$ of a set of elements represents the **linear order** $\{(x_i, x_j) | i \leq j\}$. Linear orders are the special case of a partial order whose comparability graph is complete. Every poset relation can be expressed as the intersection of a set of linear orders [5]. The **dimension** of a poset relation is the minimum number of linear orders such that their intersection is the poset relation.

The comparability graphs of two-dimensional poset relations are known as **permutation graphs**. For $k > 2$, it is NP-complete to determine whether a given poset relation has dimension k [36], but polynomial-time algorithms are known for two dimensions.

Theorem 1.1 *[31] A graph G is a permutation graph iff G and its complement are comparability graphs.*

The simple idea behind the proof is the following. If G is a permutation, then a transitive orientation of G can be represented by the intersection of two linear orders. The intersection of one of these orders and the reverse of the other gives a transitive orientation of the complement, hence the complement of G is a comparability graph. Conversely, if G and its complement are comparability graphs et D_1 and D_2 denote transitive orientations of G and let $(D_2)^T$ denote the transitive orientation obtained by reversing the directions of all edges of D_2. It is easy to see that $D_1 \cup D_2$ is a linear order L_1, and that $D_1 \cup (D_2)^T$ is a linear order L_2. $D_1 = L_1 \cap L_2$, so D_1 is two-dimensional, and G is a permutation graph (see Figures 4 and 5).

This theorem is the basis of the linear time bound for recognizing permutation graphs given in [21]: the algorithm finds orientations D_1 and D_2 of G and its complement that are transitive if both of these graphs are comparability graphs. It then computes vertex orderings corresponding to $D_1 \cup D_2$ and $D_1 \cup (D_2)^T$, and checks whether their intersection is, in fact, D_1.

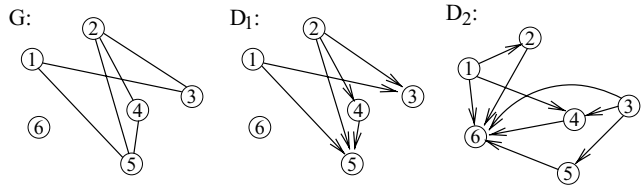


Figure 4. A graph G, transitive orientation D_1 of G, and a transitive orientation D_2 of its complement.

Let I be a set of intervals on the real line such that no two endpoints of intervals coincide. Let G_n be the graph of non-intersecting pairs of intervals, let G_c be the graph of pairs of intervals such that one is a subset of the other, and let G_1 be the graph of pairs of intervals where each contains one endpoint of the other. Let us use double subscripts to denote unions of these graphs. For instance G_{1n} denotes $G_1 \cup G_n$. The interval graph represented by the intervals is G_{1c}.

G_n, G_c, and G_1 give a partition of the complete graph. We have observed above that G_n, which is the complement of the corresponding interval graph, is a comparability graph. Also, the subset relation among members of I is transitive, so G_c is also a comparability graph. In fact, G_c is a permutation graph: listing the vertices in right-to-left order of left endpoint and in left-to-right order of right endpoint gives the corresponding linear orders. Thus, its complement, $G_1 \cup G_n$, is also a permutation graph.

If $G_1 \cup G_n$ is a permutation graph, then it has a transitive orientation. Another way to see this is to let I be a set of intervals that represent G_n, G_1, and G_c. If xy is an edge of $G_n \cup G_1$, then orient xy from x to y if the the left endpoint of interval x

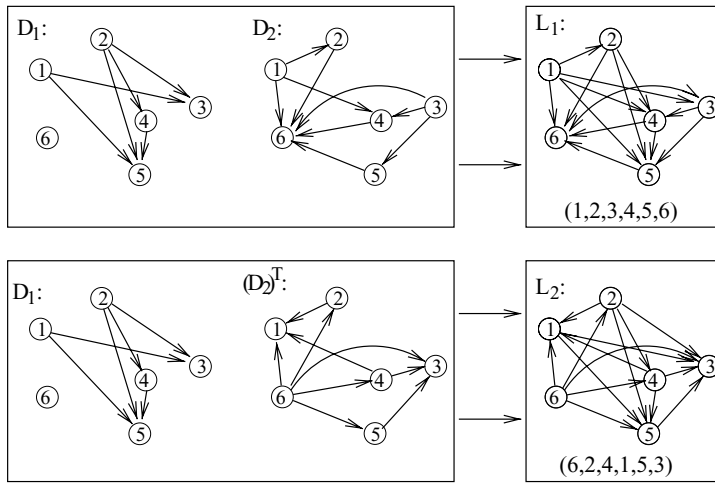


Figure 5. The union of D_1 and D_2 is a linear order, $L_1 = (1, 2, 3, 4, 5, 6)$. The union of D_1 and the transpose $(D_2)^T$ of D_2 is another linear order, $L_2 = (6, 2, 4, 1, 5, 3)$. Since $D_1 = L_1 \cap L_2$, D_1 is a two-dimensional partial order, and G is a comparability graph.

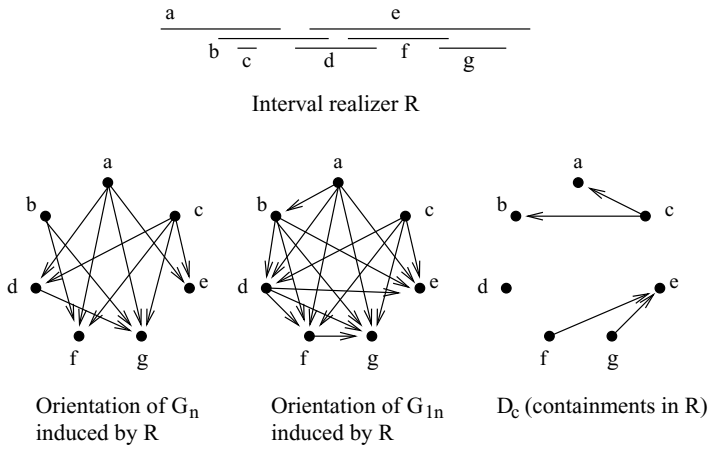


Figure 6. An interval realizer of an intersection matrix yields an **interval orientation** of G_{1n}: edges are oriented from earlier to later intervals. An interval orientation is transitive, and its restriction to G_n is also transitive. The union of D_c and an interval orientation is a linear order, and gives the order of right endpoints in the realizer. Similarly, the union of $(D_c)^T$ and the interval orientation gives the order of left endpoints.

is to the left of the left endpoint of interval y. It is easy to see that this is a transitive orientation D_{1n} of G_{1n}. This is illustrated in Figure 6.

$D_{1n} \cup D_c$ is the linear order, and it is not hard to see that it gives the order of right endpoints in I. Similarly, $D_{1n} \cup (D_c)^T$ gives the order of left endpoints of I.

The interval graph of I fails to capture some of these embedded comparability graphs. Thus, it is natural to use a matrix that tells, for each pair of intervals, the type of intersection (G_n, G_c, or G_1) that they share. We will see below how this matrix, its embedded comparability graphs, and their transitive orientations, are a key element in the linear time bound for recognition of circular-arc graphs.

An algorithm for a decision problem or optimization problem that provides no accompanying documentation of its decision are of theoretical value, but not very useful in practice. The reason is that actual implementations often have bugs. If the underlying algorithm does not provide a certificate of the correctness of its determination, a user, or a programmer, has no way to determine whether its determination is correct, or the result of a bug.

Failure to employ a certifying algorithm is illustrated by a lesson from the development of the LEDA package [27]. One of the more important utilities of that package is one for producing planar embeddings of graphs, owing to the applications of this problem in VLSI design. The procedure produced a certificate (the planar embedding) when the input graph was planar, but provided no supporting evidence when the input graph was not planar, allowing it to make false negative judgments that went undetected for some time after its release [26]. The problem was amended by changing the

underlying algorithm so that it returned one of the forbidden subgraphs of planar graphs ($K_{3,3}$ or K_5), which were described by Kuratowski in a famous theorem [15]. See [35] and [27, section 2.14] for general discussions on result checking.

In view of the practical importance of certificates, it is surprising that the theory community has often ignored the question of requiring an algorithm to support its output, even in cases when the existence of adequate certificates, such as Kuratowski's subgraphs, is well-known. One consequence of the results described below is that they yield suitable certificates for certain well-known decision problems, and are the basis of certificate-producing algorithms for recognizing interval graphs and permutation graphs [14], as well as for recognizing so-called **consecutive-ones matrices**.

An interesting type of certificate is the one described above for coloring comparability graphs. There is no known linear-time algorithm for recognizing comparability graphs. If the input graph is a comparability graph, the algorithm returns the coloring and a clique of the same size to prove that the coloring is a minimum one. If the graph is not a comparability graph, one of two things can happen: the algorithm returns a certificate that it is not a comparability graph, or it returns a minimum coloring and the clique certificate. Thus, if it returns a minimum coloring, one can be certain that the coloring is correct, but not that the input graph is a comparability graph. On the other hand, if it returns the certificate that the input graph is not a comparability graph, one can be certain of this, but not know how to color it.

Section 3 reviews properties of the well-known modular decomposition and its dual Gamma relation. Another excellent reference is a survey by Möhring's [28]. Section 4 describes a similar decomposition and forcing relation on labeled matrices that gave rise to the linear time bound for recognition of circular-arc graphs [23]. Section 5 describes a similar decomposition on arbitrary 0-1 matrices, whose dual forcing relation gives a certifying algorithm for the consecutive-ones property [18], and yet another that is closely related to that of Section 4, which led to an $O(n + m \log n)$ algorithm for recognizing probe interval graphs [25].

2. Preliminaries

If G is a graph, let $V(G)$ denote its vertex set, $n(G)$ denote the number of vertices, $E(G)$ denote the edge set, and $m(G)$ denote the number of edges. When G is understand, we may denote these by V, n, E, and m. If $X \subseteq V(G)$, $G|X$ denotes the subgraph of G induced by X, namely, the result of deleting vertices of $V(G) - X$, together with their incident edges.

Unless otherwise indicated, the term **graph** will mean an undirected graph. Let $\overline{G}$ denote the complement of a graph G. We may view an undirected graph as a special case of a (symmetric) directed graph. Each undirected edge xy consists of a pair of **twin** directed edges (x, y) and (y, x). Thus, we may speak of the directed edges in an undirected graph.

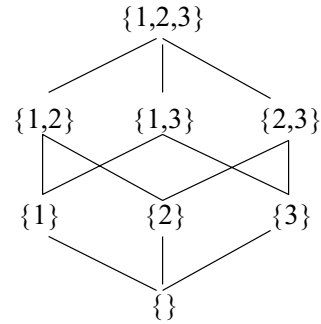


Figure 7. The Hasse diagram of the family of subsets of {1, 2, 3}.

A graph G is **bipartite** if there exists a partition X, Y of its vertices such that every edge of G has an edge in both sets. A graph is bipartite if and only if it has no cycle of odd length [4].

A **clique** in a graph is a set of vertices that are pairwise adjacent. A clique is **maximal** if no vertices can be added to it to turn it into a larger clique, and a **maximum clique** if there is no clique of larger size in the graph.

A poset relation can be represented with a **Hasse diagram**. This is obtained by starting with the directed-graph representation of the poset, and deleting all transitive edges, since they are implied by the remaining edges. By convention, the directions of the remaining edges are depicted not with arrowheads, but by arranging the nodes in a diagram so that all of the edges are directed upward. Figure 7 illustrates the Hasse diagram of the subset relation among the members of a family of sets.

A **linear extension** of a poset relation R is a linear order L such that $R \subseteq L$. We have seen that a poset relation is a transitive, antisymmetric, and reflexive relation. An **equivalence relation** is a transitive, symmetric, and reflexive relation. A relation is an equivalence relation if and only if there is a partition of the elements into **equivalence classes** such that the relation is exactly those ordered pairs of elements that reside in a common equivalence class.

If M is a matrix and A is a set of rows of M, then $M[A]$ denotes the **submatrix induced by** A, namely, the result of deleting all rows not in A. Similarly, if B is a set of columns, then $M[B]$ denotes the submatrix induced by B. If M is an $n \times n$ matrix and $X \subseteq \{1, 2, \ldots, n\}$, then $M|X$ denotes the submatrix that results from deleting rows not in X and columns not in X.

3. Graph Modules and The Γ Relation

Suppose ab and bc are two edges of a comparability graph G such that a and c are nonadjacent. In any transitive orientation F of G, exactly one of (a, b) and (b, a) will

appear. Similarly, one of (b, c) and (c, b) will appear in F. However, the fact that a and c are nonadjacent allows us to say more: if (a, b) appears in F then (b, c) *doesn't* appear, which means that (c, b) *must* appear in F. Thus, (a, b) and (b, c) are **incompatible orientations**, since they can't appear together in any transitive orientation. Similarly, (b, a) and (c, b) are incompatible. This defines an **incompatibility relation** among directed edges of G.

We can capture the incompatibilities with the following graph:

Definition 3.1 *The* Γ**-incompatibility graph** *of an undirected graph* $G = (V, E)$ *is the undirected graph* $I_\Gamma(G)$ *that is defined as follows:*

- *The vertex set of* $I_\Gamma(G)$ *is the set* $\{(x, y)|xy \in E\}$ *of directed edges of* G.
- *The edge set of* $I_\Gamma(G)$ *consists of the elements of the following sets:*
  - $\{\{(a, b), (b, a)\}|ab$ *is an edge of* $G\}$*;*
  - $\{\{(a, b), (b, c)\}|ab, bc$ *are edges of* G *and* ac *is not an edge of* $G\}$.

An example is given in Figure 8. Note that $I_\Gamma(G)$ is defined whether or not G is a comparability graph. An orientation of G is a selection of half of the directed edges of G, one from each twin pair $\{(a, b), (b, a)\}$. Clearly, such an orientation can only be transitive if this selection is an independent set in $I_\Gamma(G)$. The reverse of a transitive orientation selects the complement of this set, and is also a transitive orientation. Therefore, G can only be a comparability graph if $I_\Gamma(G)$ is bipartite. We will see below

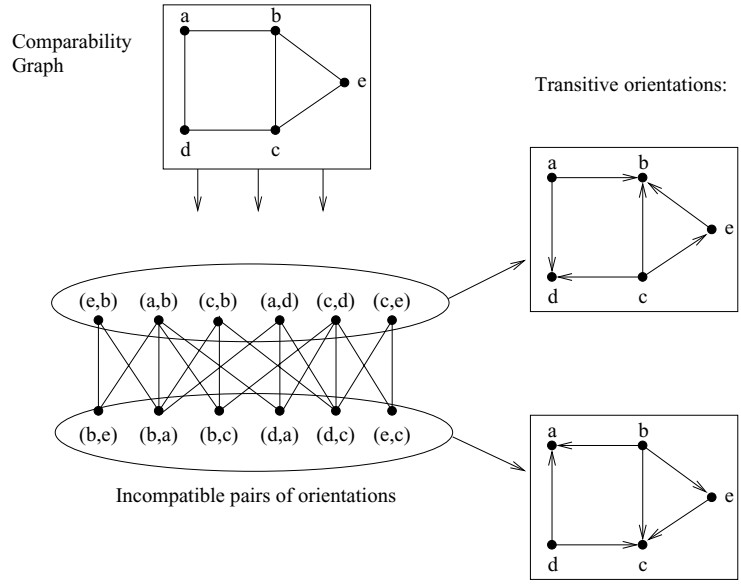


Figure 8. The incompatibility graph on directed edges of an undirected graph G. A graph is a comparability graph iff its incompatibility graph is bipartite; in this case, each transitive orientation is a bipartition class of a bipartition of the incompatibility graph.

that this condition is also sufficient. Therefore, an odd cycle in $I_\Gamma(G)$ can serve as a certificate that G is not a comparability graph.

Definition 3.2 *In an undirected graph G, let us say that two vertices x and y are* **even-equivalent** *if there is a walk of even length from x to y.*

Even-equivalence is clearly an equivalence relation. If G is a connected graph, it has two even-equivalence classes if and only if it is bipartite. If it is not bipartite, it has only one even-equivalence class. If G is a connected graph, it suffices to show that two vertices are even-equivalent in order to establish that they belong in the same bipartition class if G is later discovered to be bipartite. (We are interested in the even-equivalence in $I_\Gamma(G)$, and, since its size is non-linear in the size of G, we cannot find out in linear time whether it is bipartite in a straightforward way.)

For carrying out proofs, we will sometimes find it useful to use a relation, called Γ, to denote a special case of even-equivalence. This relation was first described by Gallai [8]. If ab and bc are two edges in an undirected graph G such that ac is not an edge, then $(a, b)\Gamma(c, b)$ and $(b, a)\Gamma(b, c)$. Note that in this case, $((a, b), (b, c), (c, b))$ is an even-length path from (a, b) to (c, b) and $((b, a), (c, b), (b, c))$ is an even-length path from (b, a) to (b, c). Thus, $e_1 \Gamma e_2$ implies that e_1 and e_2 are even-equivalent.

The incompatibility relation above is only a restatement of the Γ relation that makes it easier to discuss certain concepts of interest to us here. For instance, an odd cycle in $I_\Gamma(G)$ gives a certificate that G is not a comparability graph, and it is harder to describe the analog of this odd cycle in the Γ relation. What are commonly called the "color classes" in the Γ relation are the connected components of $I_\Gamma(G)$, and what are called the "implication classes" are the even-equivalence classes of the components.

3.1. Modular Decomposition and Rooted Set Families

A **module** of an undirected graph $G = (V, E)$ is a set X of vertices that all have identical neighborhoods in $V - X$. That is, for each $y \in V - X$, y is either adjacent to every member of X or to none of them (see Figure 9).

Note that V and its one-element subsets are modules for trivial reasons. These are the **trivial modules** of G. Graphs that have only trivial modules are **prime**.

Definition 3.3 *Let us say that two sets X and Y* **overlap** *if X and Y intersect, and neither of the two is a subset of the other.*

Definition 3.4 *[30] Let $\mathcal{F}$ be a family of subsets of a universe V. Then $\mathcal{F}$ is a* **rooted set family** *if and only if it satisfies the following properties:*

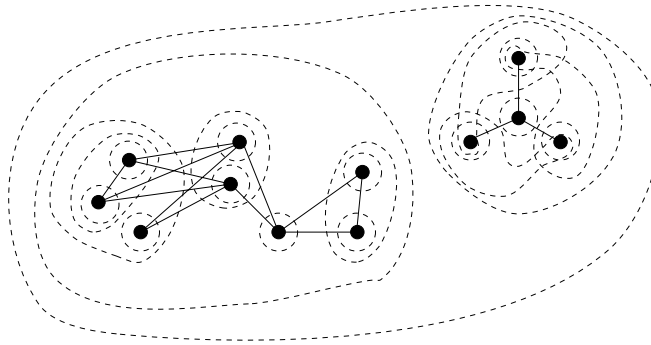


Figure 9. A graph and its modules. A module is a set X of vertices such that for each $y \in V(G) - X$, either y is adjacent to every element of X, or y is nonadjacent to every element of X.

- *All members of $\mathcal{F}$ are non-empty;*
- *$V \in \mathcal{F}$;*
- *$\{x\} \in \mathcal{F}$ for each $x \in V$;*
- *Whenever X and Y are overlapping members of $\mathcal{F}$, then $X \cup Y$, $X \cap Y$, and $(X - Y) \cup (Y - X)$ are all members of $\mathcal{F}$.*

Below, we will see that the modules of a graph constitute a rooted set family, and that rooted set families have an elegant and compact representation with a tree. The term "rooted set family" contrasts with a closely related class of set families, called "unrooted set families" that can be represented with an unrooted tree. Unrooted set families are not of concern here; a treatment can be found in [12].

Lemma 3.5 *If X and Y are overlapping members of a rooted set family, then $X - Y$ and $Y - X$ are also members of the rooted set family.*

Proof. Since X and Y are overlapping, $(X - Y) \cup (Y - X)$ is a member of $\mathcal{F}$. Since X and $(X - Y) \cup (Y - X)$ are also overlapping members of $\mathcal{F}$, their intersection, $X - Y$, is also a member. By symmetry, $Y - X$ is also a member. Q.E.D.

Theorem 3.6 *The modules of a graph G are a rooted set family on universe $V(G)$.*

Proof. $V(G)$ and its single-element subsets are modules for trivial reasons.

Suppose X and Y are overlapping modules. To show that $X \cup Y$ is a module, we let z be an arbitrary vertex in $V(G) - (X \cup Y)$ and show that it has a uniform relationship to all members of $X \cup Y$. This is true if z has no neighbor in $X \cup Y$. Otherwise, suppose without loss of generality that it has a neighbor in X. Since X is a module and $z \in V(G) - X$, all members of X are neighbors of z. This includes some vertex $w \in X \cap Y$. Since $w \in Y$, Y is a module, and $z \in V(G) - Y$, all members of Y are neighbors of x.

This establishes that if $X \cap Y$ fails to be a module, it is because some vertex $z \in (X - Y) \cup (Y - X)$ has a non-uniform relationship to members of $X \cap Y$, and that if $(X - Y) \cup (Y - X)$ fails to be a module, it is because some $z' \in X \cap Y$ has a non-uniform relationship to members of $X \Delta Y$. This can be ruled out if there is no edge with endpoints in both $X \cap Y$ and $(X - Y) \cup (Y - X)$. Otherwise, let uv be an edge of G with $u \in X \cap Y$ and $v \in (X - Y) \cup (Y - X)$. Without loss of generality, suppose that $v \in X - Y$. Since Y is a module, v is adjacent to every member of Y. Since X is a module and $v \in X$, every member of $Y - X$ is a neighbor of every member of X. Since Y is a module, every member of $X - Y$ is a neighbor of every member of Y. We conclude that each member of $(X - Y) \cup (Y - X)$ is a neighbor of every member of $X \cap Y$, so z and z' cannot exist. Q.E.D.

Definition 3.7 *Let $\mathcal{F}$ be a rooted set family. The* **strong members** *of $\mathcal{F}$ are those that overlap with no other member of $\mathcal{F}$.*

Lemma 3.8 *The Hasse diagram of the subset relation on strong members of a rooted set family is a tree.*

Proof. There is a unique maximal node, namely, $V(G)$. Thus, the Hasse diagram is connected. Suppose there is an undirected cycle in the Hasse diagram. The cycle has a minimal element X. The two edges of the cycle extend upward to two elements Y and Z. If $Y \subset Z$, then there is no direct edge in the Hasse diagram from X to Z, a contradiction. Similarly, $Z \not\subset Y$. However, $Y \cap Z$ is nonempty, since it contains X. Thus, Y and Z are overlapping, contradicting their status as strong members of $\mathcal{F}$. Q.E.D.

If $\mathcal{F}$ is a rooted set family, let us therefore denote the Hasse diagram of its strong members by $\mathcal{T}(\mathcal{F})$.

Lemma 3.9 *If $\mathcal{F}$ is a rooted set family, every weak member of $\mathcal{F}$ is a union of a set of siblings in $\mathcal{T}(\mathcal{F})$.*

Proof. Suppose a weak member X of $\mathcal{F}$ intersects two siblings, Y and Z in $\mathcal{T}(\mathcal{F})$. If $Y \not\subset X$ then X and Y are overlapping, contradicting the fact that Y is strong in $\mathcal{F}$. Thus, $Y \subset X$, and, by symmetry, $Z \subset X$. Q.E.D.

Lemma 3.10 *Let $\mathcal{F}$ be a rooted set family, let X be an internal node of $\mathcal{T}(X)$ and let $\mathcal{C}$ be its children. If some union of children of X is a weak member of $\mathcal{F}$, then the union of every subset of children of X is a member of $\mathcal{F}$.*

Proof. Claim 1: *A maximal subset of $\mathcal{C}$ whose union is a weak module must have cardinality $|\mathcal{C}| - 1$.*

Proof of Claim 1: Suppose $\mathcal{C}'$ is a maximal subset of $\mathcal{C}$ whose union is a weak module and $|\mathcal{C}'| < |\mathcal{C}| - 1$. Let $X = \bigcup \mathcal{C}'$. Since X is weak, it overlaps with some other member Y of $\mathcal{F}$. By Lemma 3.9, Y is also the union of a subset of $\mathcal{C}$. Y contains all members of $\mathcal{C} - \mathcal{C}'$; otherwise $X \cup Y$ is also weak, contradicting the maximality of

X. By Lemma 3.5, $Y - X$ is a member of $\mathcal{F}$. Since this is the union of more than one member and fewer than all members of $\mathcal{C}$, it is weak. Therefore, it overlaps with some member W of $\mathcal{F}$ that is also a union of members of $\mathcal{C}$. If W contains X, it contradicts the maximality of X. If W overlaps X, then $W \cup X$ contradicts the maximality of X.

Claim 2: *Let $\mathcal{A} \subseteq \mathcal{C}$. If, for every pair $A_i, A_j \in \mathcal{A}$, $A_i \cup A_j \in \mathcal{F}$, then $\bigcup(\mathcal{A}) \in \mathcal{F}$.*

Proof of Claim 2: This is obvious from the fact that the union of overlapping members of $\mathcal{F}$ is a member of $\mathcal{F}$. Formally, the claim is trivially true if $|\mathcal{A}| \leq 2$. Suppose $|\mathcal{A}| = k > 2$ and that the claim is that it is true for unions of $k - 1$ members of $\mathcal{C}$. For any $B, C \in \mathcal{A}$, $\bigcup(\mathcal{A} - \{B\})$ is a member of $\mathcal{F}$ by induction, and $B \cup C$ is an overlapping members of $\mathcal{F}$, so their union, $\bigcup(\mathcal{A})$, is also a member of $\mathcal{F}$.

Claim 3: *For every $A \in \mathcal{C}$, $A' = \bigcup(\mathcal{C} - \{A\})$ is a member of $\mathcal{F}$, and for every $C_i, C_j \in \mathcal{C}$, $C_i \cup C_j$ is a member of $\mathcal{F}$.*

Proof of Claim 3: By Claim 1, there is some $A_1 \in \mathcal{C}$ such that $A'_1 = \bigcup(\mathcal{C} - \{A_1\}) \in \mathcal{F}$. Since A'_1 is weak, it overlaps with some member Z of $\mathcal{F}$ that contains A_1 and that is also the union of a subfamily of $\mathcal{C}$. Z is contained in some maximal member A'_2 of $\mathcal{F}$ such that $A'_2 = \bigcup(\mathcal{C} - \{A_2\})$ for some $A_2 \in \mathcal{C}$. $A_2 \neq A_1$ since A_1 is a subset of A'_2. The symmetric difference of A'_1 and A'_2 is $A_1 \cup A_2$, hence $A_1 \cup A_2$ is a member of $\mathcal{F}$.

Suppose by induction that for some k such that $2 \leq k < |\mathcal{C}|$, there is are members $A_1, A_2, \ldots, A_k \in \mathcal{C}$ such that for every A_i in this set, A'_i is a member of $\mathcal{F}$, and for every pair A_i, A_j in this set, $A_i \cup A_j$ is a member of $\mathcal{F}$. By Claim 2, $W = \bigcup(\{A_1, A_2, \ldots, A_k\})$ is a member of $\mathcal{F}$. Let $A'_{k+1} = \bigcup(\mathcal{C} - \{A_{k+1}\}) \in \mathcal{F}$ be a maximal union of members of $\mathcal{C}$ that contains W and is a member of $\mathcal{F}$. Since A'_{k+1} contains W, $A_{k+1} \notin \{A_1, A_2, \ldots, A_k\}$. For each $A_i \in \{A_1, \ldots, A_k\}$, $A_i \cup A_{k+1} = (A'_i - A'_{k+1}) \cup (A'_{k+1} - A'_i)$ is a member of $\mathcal{F}$. This concludes the proof of Claim 3. The lemma follows from Claims 2 and 3. Q.E.D.

Lemmas 3.8, 3.9, and 3.10 give the following:

Theorem 3.11 *[30] If $\mathcal{F}$ is a rooted set family, then the internal nodes of the Hasse diagram of the strong members can be labeled as* **prime** *and* **degenerate** *so that the members of $\mathcal{F}$ are the nodes of the tree and the unions of children of degenerate nodes.*

Let us call the labeled tree given by Theorem 3.11 $\mathcal{F}$'s **decomposition tree**. When a node has two children, the appropriate label is ambiguous; we will assume that any such node is labeled degenerate. For the special case of modules of a graph, this tree is known as the **modular decomposition** of the graph. Figure 10 gives an example. However, below, we will use the fact that such a representation exists for any rooted set family.

As a data structure, the modular decomposition can be conveniently represented in $O(n)$ space. Each leaf is labeled with the graph node that it represents. There is no need to give an explicit list of members of the set X represented by an internal

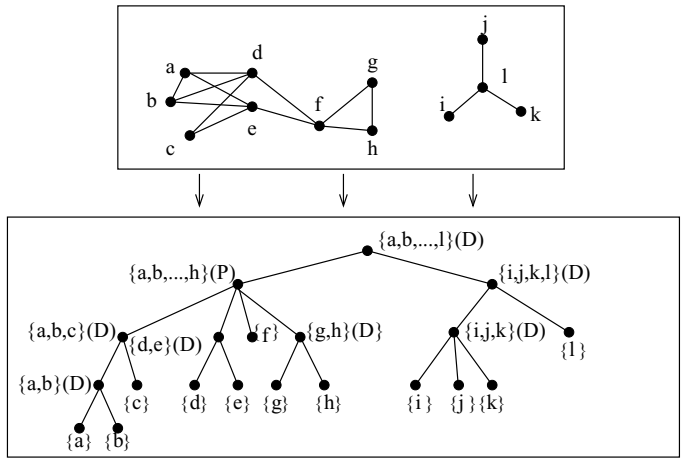


Figure 10. The modular decomposition of a graph. The nodes are labeled **prime** (P) and **degenerate** (D). Every node of the tree and every union of children of a degenerate node is a module. For instance, $\{i, j, k\}$ is a module because it is a node of the tree, and $\{i, j\}$, $\{j, k\}$, and $\{i, k\}$ are modules because they are unions of children of a degenerate node.

node, as these can be found in $O(|X|)$ time by visiting the leaf descendants of the node.

Let us call a partition of the vertices of a graph a **congruence partition** if every partition class is a module. The partition $\{\{x\}|x \in V\}$ is a trivial congruence partition. So is the partition $\{V\}$ that has only one partition class. If G is non-prime, there will be other congruence partitions.

It is easy to verify that if X and Y are disjoint modules, then $X \times Y$ is either disjoint from the edges of G or a subset of the edges of G. Thus, we may describe the relationship of X and Y as one of two types: adjacent or nonadjacent. If $\mathcal{P}$ is a congruence partition, the adjacencies and nonadjacencies among its partition classes define a **quotient graph**, $G/\mathcal{P}$, whose vertices are the members of $\mathcal{P}$ and whose edges give the adjacencies among members of $\mathcal{P}$ (Figure 11).

Let A and A' be two sets, each consisting of one vertex from each partition class in $\mathcal{P}$. $G|A$ and $G|A'$ are isomorphic to this quotient. Thus, the quotient $G/\mathcal{P}$ completely specifies those edges of G that are not subsets of one of the partition classes. The quotient $G/\mathcal{P}$, together with the **factors** $\{G|X : X \in \mathcal{P}\}$ completely specify G.

Definition 3.12 *If X is the set represented by a node of the decomposition tree, let $\mathcal{C}$ be the sets represented by its children. X's* **associated quotient graph** *is the graph $(G|X)/\mathcal{C}$.*

Lemma 3.13 *If G is an undirected graph, D is a degenerate node of its modular decomposition, and $\mathcal{C}$ is its children then the associated quotient graph $G' = (G|D)/\mathcal{C}$ is either a complete graph or the complement of a complete graph.*

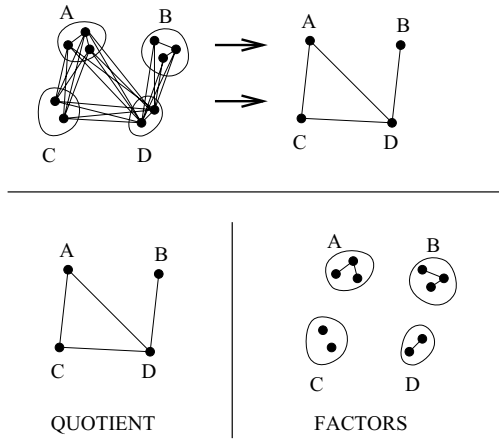


Figure 11. A quotient on the graph of Figure 9. For any pair $\{X, Y\}$ of disjoint modules, either every element of $X \times Y$ is an edge or none is. Thus, X and Y may be viewed as **adjacent** or **nonadjacent**. If $\mathcal{P}$ is a partition of the vertices of a graph where each part is a module, the adjacencies of the parts can be described by a **quotient graph** $G/\mathcal{P}$. The **factors** are the subgraphs induced by the members of $\mathcal{P}$. Given the quotient and the factors, it is easy to reconstruct G.

Proof. Since D is a module, the modular decomposition of $G|D$ is just the subtree rooted at D in the modular decomposition of G. Since the union of every nonempty subfamily of members of $\mathcal{C}$ is a module of G, every nonempty set of vertices of $G' = (G|D)/\mathcal{C}$ is a module in G'. Suppose G' is neither a complete graph nor the complement of a complete graph. Then there exists a vertex $v \in V(G')$ that has a mixture of neighbors and non-neighbors in $V(G') - \{v\}$. But then $V(G') - \{v\}$ fails to be a module, a contradiction. Q.E.D.

By Lemma 3.13, we can call a degenerate node D a **1-node** if its children are adjacent to each other, and a **0-node** if its children are non-adjacent to each other.

Lemma 3.14 *No 1-node can be a child of a 1-node, and no 0-node can be a child of a 0-node.*

Proof. Let A be a 0-node, let B be a 0-node child, and let $\mathcal{B}$ be the children of B. Then the union of any member of $\mathcal{B}$ and any child of A other than B is a module of $G|A$ that overlaps B. By the substructure rule, this is a module of G that overlaps B, contradicting B's status as a strong module. Thus, a 0-node cannot be a child of another. Since the modular decomposition of $\overline{G}$ is identical except for exchanging the roles of 0-nodes and 1-nodes, no 1-node can be a child of another. Q.E.D.

There are a number of such properties that we can show for modules, but we will find it convenient to prove them within an abstract context that will generalize easily to other structures that we discuss below. This context is an abstraction that appeared in [29] and in [30], where it was shown that modular decomposition is a special case.

The abstraction avoids explicit mention of graphs and modules, while retaining those properties required to prove most of the interesting theorems about modules. Möhring has shown that a variety of interesting structures other than modules in graphs are instances of the abstraction, so a proof that uses only the abstraction is more general than one that makes specific mention of graphs and modules. We give some others below.

When one applies the following definition, $\mathcal{S}$ corresponds to the set of all undirected graphs, $V(G)$ corresponds to a set of vertices of a graph G, $G|X$ corresponds to the subgraph induced by X, and $\mathcal{F}(G)$ corresponds to the set of modules of G.

Definition 3.15 *A* **quotient structure** *is some class $\mathcal{S}$ of structures, together with operators $V()$, $|$, and $\mathcal{F}()$ with the following properties.*

- *For $G \in \mathcal{S}$, $V(G)$ returns a set;*
- *For $X \subseteq V(G)$, the* **restriction of** *G* **to** *X, denoted $G|X$, yields an instance G' of $\mathcal{S}$ such that $V(G') = X$;*
- *$\mathcal{F}(G)$ defines a rooted set family on $V(G)$;*

Then $(\mathcal{S}, V(), \mathcal{F}(), |)$ defines a **quotient structure** *if it satisfies the following:*

- *("The Restriction Rule:") For each $Y \subseteq V(G)$ and $X \in \mathcal{F}(G)$, $X \cap Y \in \mathcal{F}(G|Y) \cup \{\{\}\}$*
- *("The Substructure Rule:") For each $Y \subseteq X \in \mathcal{F}(G)$, $Y \in \mathcal{F}(G)$ iff $Y \in \mathcal{F}(G|X)$.*
- *("The Quotient Rule:") Let $\mathcal{P}$ be a partition of $V(G)$ such that each member of $\mathcal{P}$ is a member of $\mathcal{F}(G)$, and let A be a set consisting of one element from each member of $\mathcal{P}$. Let $\mathcal{W} \subseteq \mathcal{P}$ and let B be the members of A that are in members of $\mathcal{W}$. Then $\bigcup \mathcal{W} \in \mathcal{F}(G)$ iff $B \in \mathcal{F}(G|A)$.*

Given the foregoing, following is easy to verify, and the proof is left to the reader.

Theorem 3.16 *Let $\mathcal{S}$ denote the set $\{G|G$ is an undirected graph$\}$, and for $G \in \mathcal{S}$, let $V(G)$ denote its vertices, let $\mathcal{F}(G)$ denote its modules, and for $X \subseteq V(G)$, let $G|X$ denote the subgraph of G induced by X. Then $(\mathcal{S}, V(), \mathcal{F}(), |)$ is a quotient structure.*

3.2. Modular Decomposition and Transitive Orientation

The following gives an initial hint of a relationship between the modular decomposition and the transitive orientations of a graph.

Lemma 3.17 *If M is a module of G, (a, b) is an edge such that $\{a, b\} \subseteq M$ and (c, d) is an edge such that $\{c, d\} \not\subseteq M$, then (a, b) and (c, d) are in different connected components of $I_\Gamma(G)$.*

Proof. Suppose that they are in the same component. There is a path in $I_\Gamma(G)$ from (a, b) to (c, d). There must therefore be an adjacent pair (u, v), (v, w) in $I_\Gamma(G)$ such that $\{u, v\} \subseteq M$ and $\{v, w\} \not\subseteq M$. However, then w is adjacent to $v \in M$ and nonadjacent to $u \in M$, contradicting the fact that M is a module. Q.E.D.

A graph $G = (V, E)$ is **prime** if the only modules it has are trivial modules, namely, V and its one-element subsets. The convention of calling a node of the modular decomposition prime if no union of its children is a weak module stems from the fact that, by the quotient rule, its associated quotient is a prime graph.

Let $\mathcal{P} = (X_1, X_2, \ldots, X_k)$ be a partition of the vertices of G into more than one partition class, together with a linear ordering of members of $\mathcal{P}$. A **pivot** is the following operation. While there is a non-singleton class X_i that is not a module, select a **pivot vertex** $z \in V - X_i$ that has both neighbors and non-neighbors in X_i. Since X_i is not a module, z exists. Let j be the index of the partition class X_j that contains z. If $j > i$, then refine the partition by changing the partition and ordering to $(X_1, X_2, \ldots, X_{i-1}, X_i \cap N(z), X_i - N(z), X_{i+1}, \ldots, X_k)$. Otherwise change it to $(X_1, X_2, \ldots, X_{i-1}, X_i - N(z), X_i \cap N(z), X_{i+1}, \ldots, X_k)$. Return the partition and its ordering when all partition classes are modules.

The refinement procedure is illustrated in Figure 12. Let us call this operation `Refine(`$\mathcal{P}$`)`. Note that when G is a prime, every non-singleton partition class X_i is

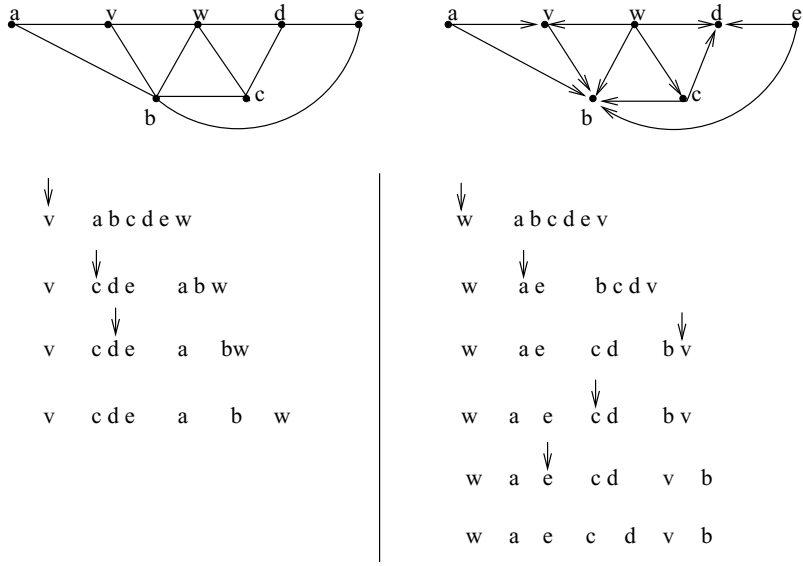


Figure 12. Transitively orienting a prime comparability graph with Algorithm 3.20. The algorithm starts by picking an arbitrary vertex v, and refining the ordered partition $(V - \{v\}, \{v\})$ using pivots. A pivot consists of selecting a vertex and splitting one of the classes that doesn't contain the pivot into neighbors and non-neighbors. The non-neighbors are placed on nearer the pivot than the neighbors. When the first partition class is a singleton set $\{w\}$, w is identified as a source in a transitive orientation. A separate refinement on $(\{w\}, V - \{w\})$ produces a linear extension of a transitive orientation.

a non-module, hence there exists a pivot vertex that can split the class. This gives the following:

Lemma 3.18 *When $\mathcal{P}$ is a linearly-ordered partition of vertices of a prime graph G where $\mathcal{P}$ has more than one partition class, then* `Refine` $(\mathcal{P}, G)$ *returns a linear ordering of the vertices of G.*

The key insight into the usefulness of the pivot operation is the following observation.

Observation 3.19 *If, after a pivot on z, $x \in X_i \cap N(z)$, then for every edge (x, u) of G that is directed from x to $X_i - N(z)$, $(x, u)\Gamma(x, z)$, and $(u, x)\Gamma(z, x)$. In particular, if all edges of G directed from $X_i \cap N(z)$ to z are even-equivalent in $I_\Gamma(G)$, then so are all edges of G that are directed from $X_i \cap N(z)$ to $X_i - N(z)$.*

The following is an algorithm for transitively orienting a prime comparability graph that first appeared in [19]. It has been used in a number of subsequent papers have used it to obtain a variety of results [20, 21, 22, 10, 23]. Instead of explicitly orienting the edges, the algorithm returns an ordering of the vertices that is a linear extension of a transitive orientation of G if G is a comparability graph.

Algorithm 3.20 *Produce an ordering of vertices of a graph G that is a linear extension of a transitive orientation if G is a comparability graph.*

Let v be an arbitrary vertex;
$\mathcal{P} := (V(G) - \{v\}, \{v\})$*;*
Call `Refine` $(\mathcal{P})$*;*
Let w be the vertex in the leftmost set returned by the call;
$\mathcal{Q} := (\{w\}, V(G) - \{w\}))$
Call `Refine` $(\mathcal{Q})$
Return the resulting linear ordering of the vertices

An illustration is given in Figure 12.

Lemma 3.21 *Let G be a prime graph, and let $(v_1, v_2, \ldots, v_n)$ be an ordering of its vertices returned by Algorithm 3.20. Then the edges of G that are directed from earlier to later vertices in this ordering are even-equivalent in $I_\Gamma(G)$.*

Proof. Let $V - \{v\} = W_1, W_2, \ldots, W_k = \{w\}$ be the sequence of sets that are leftmost classes as $\mathcal{P}$ is refined. We show that by induction on i that all edges directed from w to $V(G) - W_i$ are even-equivalent. This will imply that w is a source in a transitive orientation of G if G is a comparability graph.

Note that w is adjacent to v, since it is moved to the leftmost partition class after the initial pivot on w. Thus, for $i = 1$, the sole edge from w to $V(G) - W_1$ is in a single even-equivalence class.

Suppose $i > 1$ and all edges from w to $V - W_{i-1}$ are even-equivalent. By Observation 3.19, all edges from w to $W_{i-1} - W_i$ are even-equivalent to (w, z), where z is the pivot that splits W_{i-1}. Since (w, z) is even-equivalent to all other edges directed from w to $V - W_{i-1}$, the claim follows.

We conclude the edges directed out of w are all even-equivalent. Let us now use this to show that during the refinement of $\mathcal{Q}$, the set of edges that are directed from a vertex in a class to a vertex in a later class are even-equivalent. This is true initially for $\mathcal{Q} = (\{w\}, V - \{w\})$ by our choice of w. It remains true during partitioning by Observation 3.19, the rule for ordering partition classes, and induction on the number of pivots that have been performed. Q.E.D.

Corollary 3.22 *If G is a comparability graph, then the ordering of vertices returned by Algorithm 3.20 is a linear extension of a transitive orientation.*

Corollary 3.23 *If a graph G is prime, then $I_\Gamma(G)$ is connected.*

A straightforward $O(n + m \log n)$ implementations of Algorithm 3.20 is given in [19, 21]. A generalization which does not require G to be prime is given in [22, 10]. This can be used to compute the modular decomposition or to transitively orient a comparability graph if it is not prime. A linear-time implementation of Algorithm 3.20, together with the modular-decomposition algorithm of [19, 21], is the basis of the linear-time bound for the transitive orientation problem given in [21].

Lemma 3.24 *Let G be an undirected graph, D be a degenerate 1-node of the modular decomposition, and X and Y be two children of D. Then $X \times Y \cup Y \times X$ is a component of $I_\Gamma(G)$, and the members of $X \times Y$ are even-equivalent.*

Proof. By Lemma 3.17, no component can contain any of these edges and any edges outside of this set. It remains to show that the members of $X \times Y$ are even-equivalent, as this will also imply that $X \times Y \cup Y \times X$ is connected.

By Lemma 3.14, $\overline{G}|X$ and $\overline{G}|Y$ are connected. Let $y \in Y$, and let $x_1, x_2 \in X$ be two vertices that are adjacent in $\overline{G}|X$. Then $(x_1, y)\Gamma(x_2, y)$, hence these are even-equivalent. Since $\overline{G}|X$ is connected, the edges of $X \times \{y\}$ are even-equivalent. Similarly, for any $x \in X$, the members of $\{x\} \times Y$ are even-equivalent.

Let (a, b), (c, d) be arbitrary edges such that $a, b \in X$ and $c, d \in Y$. By the foregoing, (a, b) and (a, d) are even-equivalent and so are (a, d) and (c, d). Transitively, (a, b) and (c, d) are even-equivalent. □

Lemma 3.25 *If G is an undirected graph and P is a prime node of the modular decomposition and $\mathcal{C}$ is its children, then:*

1. *The set of edges that have endpoints in more than one member of $\mathcal{C}$ are a connected component of $I_\Gamma(G)$;*

2. *There exists an ordering of members of $\mathcal{C}$ such that the edges directed from earlier to later members in this ordering are even-equivalent.*

Proof. By Lemma 3.17 and the fact that P and its children are modules, there can be no other edges in a component that contains any of these edges.

If A is a set consisting of one representative from each member of $\mathcal{C}$, then by the quotient and substructure rules, $G|A$ is isomorphic to $(G|P)/\mathcal{C}$, and this graph is prime. By Lemma 3.23, $I_\Gamma(G|A)$ is connected, and there exists an ordering of members of A such that the edges directed from earlier to later elements are even-equivalent. We show that the analogous ordering the members of $\mathcal{C}$ satisfies the lemma.

Edges of $G|A$ consist of one representative from each member of $\{X \times Y : X, Y$ are two adjacent members of $\mathcal{C}\}$. Thus, it suffices to show that for each pair X, Y of children of $\mathcal{P}$, the members of $X \times Y$ are even-equivalent.

Let $x \in X$ and $y_1, y_2 \in Y$. Let A_1 consist of one representative vertex from each child of P, where x and y_1 are the representatives of X and Y, respectively. Let $A_2 = (A_1 - \{y_1\}) \cup \{y_2\}$. $G|A_1$ and $G|A_2$ are isomorphic to $(G|P)/\mathcal{C}$, hence they are isomorphic to each other.

The isomorphism maps y_1 to y_2 and the other vertices to themselves. It maps each edge of $G|A_1$ to itself, except for those edges incident to y_1. Since $G|A_1$ is prime, not all of its edges are incident to y_1, so some edge e is mapped to itself. This is an isomorphism from $I_\Gamma(G|A_1)$ to $I_\Gamma(G|A_2)$.

Since $I_\Gamma(G|A_1)$ is connected, there is a path in it from (x, y_1) to e, and an isomorphic path (of the same length) in $I_\Gamma(G|A_2)$ from (x, y_2) to e. Thus, (x, y_1) and (x, y_2) are even-equivalent. By symmetry, the foregoing also applies to (y_1, x) and (y_2, x).

Now, let (a, b) and (c, d) be two arbitrary edges in $X \times Y$. By the foregoing, (a, b) and (a, d) are even-equivalent, and (a, d) and (c, d) are even-equivalent. Transitively, (a, b) and (c, d) are even-equivalent. Since (a, b) and (c, d) are arbitrary members of $X \times Y$, all members of $X \times Y$ are even-equivalent. Q.E.D.

The following theorem, which summarizes the foregoing, shows that the modular decomposition gives a way to model the connected components of $I_\Gamma(G)$ compactly, and, if G is a comparability graph, to model its transitive orientations.

Theorem 3.26 *[8, 9] Let G be an arbitrary graph.*

1. *If X and Y are two children of a degenerate node that are adjacent in G, then $(X \times Y) \cup (Y \times X)$ is a connected component of $I_\Gamma(G)$. If G is a comparability graph, then $\{X \times Y, Y \times X\}$ is its bipartition.*
2. *If $X_1, X_2, \ldots, X_k$ are children of a prime node, then the set $E_Z = \{(a, b) : ab$ is an edge of G and $x \in X_i, y \in X_j, i \neq j\}$ is a connected component of $I_\Gamma(G)$. If*

G is a comparability graph, then there exists an ordering $X_{\pi(1)}, X_{\pi(2)}, \ldots, X_{\pi(k)}$ such that the elements of E_Z that go from an earlier child to a later child are one bipartition class and those that go from a later child to an earlier are another.

3. *There are no other connected components of $I_\Gamma(G)$.*

Corollary 3.27 *An undirected graph G is a comparability graph iff $I_\Gamma(G)$ is bipartite.*

Proof. A transitive orientation is a set consisting of half the vertices of $I_\Gamma(G)$ that must be an independent set. Reversing the the edges selects the complement of this set, and since the reverse of a transitive orientation is also transitive, this set is an independent set also. Thus, if G is a comparability graph, $I_\Gamma(G)$ is bipartite.

Suppose $I_\Gamma(G)$ is bipartite. Then each component is bipartite. At each prime node, assign an order to the children where one bipartition class consists of edges that are directed to the right in this order. At each degenerate node, assign an arbitrary order to the children. This induces an ordering of the leaves where every component of $I_\Gamma(G)$ has a bipartition class consisting of edges that are directed to the right in this order. Selecting this bipartition class from each component yields an acyclic orientation of G, and since it is an independent set in $I_\Gamma(G)$, it is also transitive. Q.E.D.

One might be tempted to think that the diameter of a component of $I_\Gamma(G)$ is at most n. However, this is not true. An example of this can be obtained as follows. Let n be odd, and let G_1 be the graph on $n-2$ vertices that is the complement $\overline{(P_{n-2})}$ of the graph consisting of a path $P_{n-2} = (v_1, v_2, \ldots, v_{n-2})$. Add a new vertex x that is adjacent to all vertices of G_1 to obtain a graph G_2. The path $((x, v_1), (v_2, x), (x, v_3), \ldots, (x, v_{n-2}))$ is a shortest path in $I_\Gamma(G_2)$. The path $((v_{n-2}, v_1), (v_1, v_{n-3}), (v_{n-4}, v_1), \ldots, (v_1, v_3))$ is also a shortest path in $I_\Gamma(G)$. Since $\{v_1, v_2, \ldots, v_{n-2}\}$ is a module of G_2, these are in different components of $I_\Gamma(G)$. Add a new vertex y adjacent only to v_{n-2}. This gives a graph G on n vertices, and joins these paths together with a new vertex (v_{n-2}, y) to create a shortest path in $I_\Gamma(G)$ that is longer than n.

Nevertheless, the diameter of a component of $I_\Gamma(G)$ is $O(n)$. If the component is the set of edges that go between children of a 1-node, this is obvious from the proof of Lemma 3.24. When G is prime, the diameter of $I_\Gamma(G)$ is $O(n)$, which can be confirmed from the proof of Lemma 3.21 and the observation that $O(n)$ pivots are required. Applying this result to the proof of Lemma 3.25 shows that this is also true for a component of $I_\Gamma(G)$ of a non-prime graph G, where the component consists of edges that connect children of a prime node.

Since each component has a spanning tree of diameter at most $3n-1$, then in a component that is not bipartite, there is an edge between two elements whose distance from each other in this tree has even parity. This edge and the path of tree edges between them yield an odd cycle of length $O(n)$.

By a somewhat more careful analysis of this argument, the following is derived in [14]:

Lemma 3.28 *If G is an undirected graph that is not a comparability graph, then $I_\Gamma(G)$ has an odd cycle of length at most $3n$.*

The paper gives a linear-time algorithm for returning an odd cycle of length at most $3n$ in $I_\Gamma(G)$ or $I_\Gamma(\overline{G})$ if G is not a permutation graph. This provides a certificate of the claim that G is not a permutation graph.

Such a certificate exists for graphs that are not comparability graphs. However, at present, no linear-time algorithm is known for finding one. Given a pair of incompatible edges in the orientation produced by Algorithm 3.20, one can be found in linear time. Such a pair can be found by squaring the boolean adjacency matrix for the oriented graph, but this does not give a linear time bound. This is the bottleneck in the time bound for recognizing comparability graphs.

4. Intersection Matrices

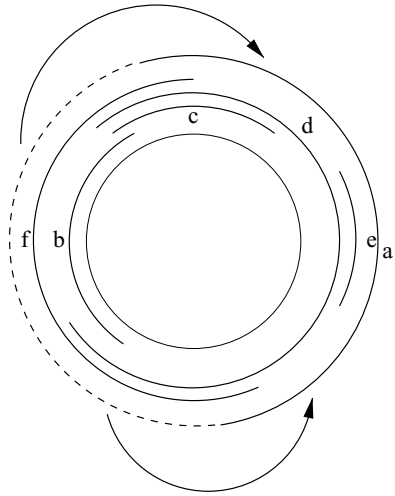
In this section, we give a forcing relation and a corresponding quotient structure that were instrumental in obtaining the linear time bound of [17] for recognizing circular-arc graphs.

Tucker gave an $O(n^3)$ algorithm [34]. Hsu improved this bound to $O(nm)$ [11], and Eschen and Spinrad further improved it to $O(n^2)$ [6]. The author recently improved it to $O(n + m)$ [17].

One reason that the problem is harder on circular-arc graphs is that there are two ways to travel between a pair of points on a circle, and only one on a line. This introduces the need for an algorithm to make choices that are not required in the interval-graph recognition problem. Also, in an interval graph the maximal cliques correspond to regions of maximal overlap among the intervals, and there are therefore $O(n)$ maximal cliques. This plays an important role in Booth and Lueker's algorithm. The number of maximal cliques in a circular-arc graph can be exponential in n [34].

To manage these problems, the algorithm of [17] works with an **intersection matrix** T, rather than directly with the graph. Like the graph, the intersection matrix tells which pairs of arcs intersect. However, it gives additional information in the case where two arcs intersect, namely, the **type** of intersection. The intersection types are classified as follows:

- **single overlap ('1'):** Arc x contains a single endpoint of arc y;
- **double overlap ('2'):** x and y jointly cover the circle and each contains both endpoints of the other;
- **containment ('c'):** arc x is contained in arc y;
- **transpose containment ('t'):** arc x contains arc y;
- **nonadjacency ('n'):** arc x does not intersect arc y.



	a	b	c	d	e	f
a		n	1	c	t	2
b	n		1	2	n	c
c	1	1		c	n	1
d	t	2	t		t	2
e	c	n	n	c		n
f	2	t	1	2	n	

Figure 13. A **flip** consists of rerouting an arc's path between its endpoints; in this case arc a is flipped. This swaps the roles of n and t, and 2 and c in a row, and swaps the roles of n and c, and 2 and t in a column.

This can be captured with a matrix T where the entry in row i and column j is the label from $\{1, 2, c, t, n\}$ that gives the type of relationship between arc i and arc j. For obtaining a linear time bound, the matrix is represented by labeling the edges of G with their intersection types from $\{1, 2, c, t\}$ and letting a label n be given implicitly by the absence of an edge.

The paper gives a linear-time algorithm for labeling the edges so that they are consistent with some realizer of G, even though the actual realizer is not known. Next, the algorithm makes use of a **flip** operation. This is the operation of rerouting an arc the opposite way around the circle, while keeping its endpoints (see Figure 13). The effect on G of a flip is unclear if only G is known. However, if the intersection matrix of the realizer is known, it is a simple matter to transform the matrix to reflect the intersection relationships in the new realizer, even if the actual realizer is not known. In the row corresponding to the flipped arc, this swaps the roles of n and t, and the roles of 2 and c. In the column column corresponding to the flipped arc, it swaps the roles of n and c, and the roles of 2 and t.

Without knowledge of the actual realizer, the algorithm finds the set of arcs that contain a particular point on the circle, and flips them. This results in the intersection matrix of an interval realizer; cutting the circle at this point and straightening it out into a line illustrates this. This makes the problem considerably easier, as working with interval realizers sidesteps the difficult issues mentioned above when one works with arcs around the circle. All that is required now is to find an interval realizer of this resulting matrix. One can then wrap this realizer around the circle again and invert the sequence of flips that were performed to obtain the interval realizer, but this time performing the flips on the realizer, rather than on the intersection matrix. The result is circular-arc realizer of the original intersection matrix, which is a circular-arc realizer of G.

No circular-arc realizer of G exists if G fails to be a circular-arc graph. If G is not a circular-arc graph, the algorithm either halts early or returns a circular-arc realizer that does not match G. Determining whether a proposed realizer matches G can easily be carried out in time linear in the size of G.

In this section, we describe elements of the central step of the algorithm, which is finding whether an intersection matrix has an interval realizer, and, if so, producing a realizer as a certificate. Note that every realizer of the matrix realizes G, but the converse may not be true.

A double overlap cannot occur in an interval realizer, so we may assume that these relationships are absent from the matrix. Let us therefore redefine an intersection matrix to be any matrix whose off-diagonal entries are drawn from the set $\{1, c, t, n\}$. The matrix is an **interval matrix** if it has an interval realizer.

We may assume without loss of generality that all endpoints of intervals in an interval realizer are unique, since if this is not the case, they can be perturbed to make this true without affecting the represented graph. In an interval realizer the exact positions of the endpoints on the line are irrelevant; what is important is the *order* in which the endpoints occur. Thus, we can represent an interval realizer with string whose letters are the vertices of G, and where each vertex appears twice in the string, once for its left endpoint and once for its right endpoint. We may therefore view the set of interval realizers of a graph as a language over an alphabet whose letters are the vertices of G.

Let R be an interval realizer, expressed as a string over the alphabet V, and let $X \subseteq V$. Then $R|A$ denotes the result of deleting all characters in $V - X$ from this string. That is, $R|X$ is the result of deleting all intervals except for those in X. Let $T|X$ denote R's intersection matrix. Note that $T|X$ is also the submatrix given by rows and columns in X.

In a way that is similar to the intersection matrices for interval graphs, an intersection matrix for a set of arcs defines four graphs with vertex set V. $G_n = \overline{G}$ is the graph whose edges are those pairs labeled n. Similarly, G_1, D_c and D_t are the graphs of edges labeled 1, c, and t, respectively. We use the notation D_c and D_t to emphasize that these graphs are directed graphs. As before, we let unions of these graphs are denoted with multiple subscripts, *e.g.* G_{1n} is the $G_1 \cup G_n$. Let $V(T)$ denote the vertex set on which these graphs are defined. For $v \in V(T)$, let $N_1(z)$, $N_n(z)$, and $N_c(z)$ denote the neighbors of v in G_1, G_n, and G_c, respectively. Let $G(T) = G_{1c}$. This is the graph that corresponds to the matrix if the matrix is an intersection matrix, but is defined for all intersection matrices.

Suppose xy is an edge of G_{1n} and the left endpoint of x is to the left of the left endpoint of y. Then since intervals x and y are disjoint or overlap, the right endpoint of x is to the left of the right endpoint of y. Let us then say that x **comes before** y in this case, even though x and y might intersect. This associates an orientation D_{1n} with

G_{1n}: the edge set of D_{1n} is $\{(x, y) : xy$ is an edge of G_{1n} and x comes before y in $R\}$. Let us call such an orientation an **interval orientation** (Figure 6).

Lemma 4.1 *An interval orientation D_{1n} of G_{1n} is a transitive orientation, and the subgraph D_n given by the restriction of D_{1n} to edges of G_n is also transitive.*

Proof. If $(x, y), (y, z) \in E(D_{1n})$, then interval x overlaps with or is disjoint from interval y and comes before it, and interval y overlaps with or is disjoint from interval z and comes before it. It follows that x overlaps with z or is disjoint from it. Thus, (x, z) is an edge of D_{1n}.

If $(x, y), (y, z) \in E(D_n)$, then interval x is disjoint from interval y and comes before it, and interval y is disjoint disjoint from interval z and comes before it. It follows that x is disjoint from z and comes before it. Thus, (x, z) is an edge of D_n. □

Given the interval orientation D_{1n}, and the graph D_c and its transpose D_t given by the intersection matrix, then we can find the left endpoint order in R; this is just the linear order D_{1nt}. Similarly, we can find the right endpoint order in R; this is just the linear order D_{1nc}. Given the left endpoint order and the right endpoint order in R, it is easy to interleave these two orders to get the full order of endpoints in R. This operation is left as an exercise.

Thus, finding a realizer of a given matrix T reduces to finding an interval orientation of G_{1n}. This brings the problem into the combinatorial domain of graphs and transitive orientations, and avoids the pitfalls in geometric arguments about possible arrangements of intervals on a line.

By analogy to the problem of finding a transitive orientation, let us derive an incompatibility graph $I_\Delta(T)$ to help us find an interval orientation.

Definition 4.2 *The* **Δ-incompatibility graph** $I_\Delta(T)$ *of an intersection matrix T is defined as follows:*

1. *The vertex set of $I_\Delta(T)$ is $\{(x, y) : xy$ is an edge of $G_{1n}\}$.*
2. *The edge set of $I_\Delta(T)$ is the pairs of vertices of the following forms:*
   (a) $\{\{(a, b), (b, a)\} : ab$ *is an edge of* $G_{1n}\}$*;*
   (b) $\{\{(a, b), (b, c)\} : ab, bc$ *are edges of* G_n *and* ac *is not an edge of* $G_n\}$*;*
   (c) $\{\{(a, b), (b, c)\} : ab, bc$ *are edges of* G_{1n} *and* ac *is not an edge of* $G_{1n}\}$*;*
   (d) $\{\{(a, b), (b, c)\}$ *or* $\{\{(a, b), (c, a)\}\} : ab$ *is an edge of* G_n*, and* ac *and* bc *are edges of* G_1.

By parts a, b, and c, an acyclic orientation of G_{1n} that is an independent set in $I_\Delta(T)$ must satisfy the requirements of Lemma 4.1. For condition d, note that if b overlaps a and c and a and c are nonadjacent, then a and c have to contain opposite endpoints of b. It follows that if a comes before b then it comes before c also. We show below that the interval orientations of G_{1n} are precisely those acyclic orientations of

G_{1n} that are independent sets of $I_\Delta(T)$, and that such an orientation exists iff $I_\Delta(T)$ is bipartite.

By analogy to the Γ relation, let us say that for two ordered pairs (a, b) and (b, c) of vertices, $(a, b)\Delta(c, b)$ and $(b, a)\Delta(b, c)$ if (a, b) and (b, c) are neighbors in $I_\Delta(T)$.

4.1. The Delta Modules of an Intersection Matrix

In this section, we show that for an intersection matrix T there is a tree, the Delta tree, which is analogous to the modular decomposition of a graph. It represents the components of $I_\Delta(T)$ in the same way that the modular decomposition represents the components if $I_\Gamma(G)$. When G is a comparability graph, its modular decomposition can be used to represent all of its transitive orientations. Similarly, when T is an interval matrix, the Delta tree can be used to represent the interval orientations of G_{1n} in the same way.

Definition 4.3 *Let T be an intersection matrix, and let $X \subseteq V(T)$.*

1. *X is a* **clique** *in T if it is a clique in the interval graph G_{1c} represented by T.*
2. *X is a* **module** *of T if it is a module of G_n, G_1, and G_c.*
3. *X is a* **Delta module** *if X is a module in T and it is either a clique or there are no edges of G_1 from X to $V(T) - X$.*

Let $\mathcal{F}(T)$ denote the Delta modules of T.

If T is an intersection matrix and $\mathcal{P}$ is a partition of $V(T)$ where each member of $\mathcal{P}$ is a module, we can define a quotient $T/\mathcal{P}$ on T. This is the submatrix induced by a set A consisting of one representative from each member of $\mathcal{P}$. Since the members of $\mathcal{P}$ are modules of G_n, G_1, and G_c, $G_n|A$, $G_1|A$, and $G_C|A$ are unaffected by changing the choice of representatives for A. Thus $T|A$ is unaffected by changing the choice of representatives for A and the quotient is well-defined.

Theorem 4.4 *The Delta modules of an intersection matrix are a rooted set family.*

Proof. Let X and Y be overlapping Delta modules. Then $X \cap Y$, $X \cup Y$, and $(X - Y) \cup (Y - X)$ are modules of G_n, G_1, and G_c by Theorem 3.6. It remains to show that these sets satisfy the additional requirements of Delta modules.

Suppose X and Y are cliques. Then there are edges of G_{1c} from each member of $Y - X$ to members of $X \cap Y$. Since X is a module of G_1 and G_c, there are edges of G_{1c} from each member of $Y - X$ to every member of X, and $X \cup Y$, $X \cap Y$, and $(X - Y) \cup (Y - X)$ are cliques, hence Delta modules.

Suppose $X \cup Y$ is not a clique. Then one of X and Y, say, X, is not a clique. There is no edge of G_1 from X to $V(T) - X$. Since $X \cup Y$ is a module of G_1, there is no edge of G_1 from $X \cup Y$ to $V(T) - (X \cup Y)$. $X \cup Y$ is a Delta module.

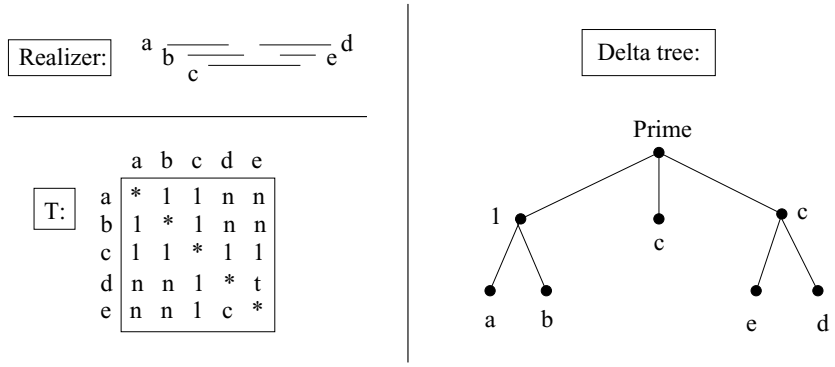


Figure 14. An interval realizer, the intersection matrix T, and the Delta tree $\Delta(T)$.

Suppose $X \cap Y$ is not a clique. Then neither X nor Y is a clique. There is no edge of G_1 from X to $V(T) - X$, or from Y to $V(T) - Y$. Thus, there is no edge of G_1 from $X \cap Y$ to $V(T) - (X \cap Y)$. $X \cap Y$ is a Delta module.

Finally, suppose that $U = (X - Y) \cup (Y - X)$ is not a clique. Then one of X and Y, say, X, is not a clique. There is no edge of G_1 from X to $V(T) - X$. Since U is a module of G_1, any edge of G_1 from U to $V(T) - U$ must go to a vertex s in $X \cap Y$. But since $(X - Y) \cup (Y - X)$ is a module, every member of $Y - X$ is a neighbor of s in G_1, and there are edges of G_1 from X to $V(T) - X$, a contradiction. Q.E.D.

Definition 4.5 *Let us call the tree decomposition of the Delta modules the* **Delta tree** *of G_{1n}, and denote it $\Delta(T)$.*

Let D be a degenerate node of the Delta tree, and let $\mathcal{C}$ be its children. The Delta modules are modules in each of G_n, G_1, and G_c, so it follows from Lemma 3.13 that $(T|D)/\mathcal{C}$ is complete in G_n, G_1, or G_c. We can call a degenerate node an n-node, 1-node, or c-node, depending on which of these cases applies. Figure 14 gives an example.

Given Theorem 4.4, the following is obvious.

Theorem 4.6 *Let $\mathcal{I}$ denote the set of all intersection matrices. then $(\mathcal{I}, V(), \mathcal{F}(), |)$ is a quotient structure.*

4.2. The Delta Tree and Interval Orientations

In this section, we show that the Delta tree has a role in interval orientations that is analogous to the role of modular decomposition in transitive orientations.

The following is analogous to Lemma 3.17.

Lemma 4.7 *If M is a Delta module of T, (a, b) is an edge such that $\{a, b\} \subseteq M$ and (c, d) is an edge such that $\{c, d\} \not\subseteq M$, then (a, b) and (c, d) are in different components of $I_\Delta(G)$.*

Proof. Suppose that they are in the same component. There is a path in $I_\Delta(G)$ from (a, b) to (c, d). There must therefore be an adjacent pair $(u, v), (v, w)$ in $I_\Delta(G)$ such that $\{u, v\} \subseteq M$ and $\{v, w\} \not\subseteq M$.

Suppose (u, v) and (v, w) fall into case (b) or case (c) of Definition 4.2. Then the relationship of w to u differs from its relationship to v. Since $u, v \in M$ and w is not, M cannot be a module, a contradiction.

Otherwise, (u, v) and (v, w) fall into case (d) of Definition 4.2. If uv is an edge of G_1 and uw is an edge of G_n, then wv is an edge of G_1, hence M fails to be a module in G_1, a contradiction. Thus, uv is an edge of G_n and uw and vw are edges of G_1. However, this means that M is not a clique, and it has outgoing edges of G_1 to w. M again fails to be a Delta module, a contradiction. Q.E.D.

Let us say that T is **prime** if it has no modules, and **Delta-prime** if it has no Delta modules. If T is prime, then it is also Delta prime, but the converse is not true.

Let us now redefine the pivot operation of Algorithm 3.20 so that we can use the algorithm to find an interval orientation, rather than a transitive orientation. As before, let $\mathcal{P} = (X_1, X_2, \ldots, X_k)$ be a partition of the vertices of G_{1n} into more than one partition class, together with a linear ordering of members of $\mathcal{P}$. A **pivot** is the following operation. Select a non-singleton class X_i. In this case, a pivot vertex is a vertex $z \in V - X_i$ such that the relationship of z to members of X_i is non-uniform. That is, in at least one of G_c, G_1, and G_n, z has both neighbors and non-neighbors in X_i.

Let j be the index of the partition class X_j that contains z. Let Y_c, Y_1, and Y_n be the members of X_i that are neighbors of z in G_c, G_1, and G_n, respectively. If $j > i$, then refine the partition by changing the partition and ordering to $(X_1, X_2, \ldots, X_{i-1}, Y_n, Y_1, Y_c, X_{j+1}, \ldots, X_k)$. Otherwise change it to $(X_1, X_2, \ldots, X_{i-1}, Y_c, Y_1, Y_n, X_{j+1}, \ldots, X_k)$. At least two of the sets X_c, X_1, and X_n are nonempty because of the choice of z. If one of the sets is empty, remove it from the partition. Let us call this type of pivot a **three-way pivot**.

There is one other type of pivot that must be used when X_i is a module that fails to be a Delta module. This is a **modular pivot**, which consists of the following operation. Let z be a vertex in $V - X_i$ such that the vertices in X_i are neighbors in G_1. Since X_i is not a Delta module, such a z exists and X_i is not a clique. Let x be a vertex with an incident edge in $G_n|X_i$, and let Y be the neighbors of x in $G_n|X_i$. We replace X_i in the ordering with $X_i - Y$ and Y, placing $X_i - Y$ first if z resides in a class that is later in the ordering than X_i, and placing it second if z resides in an earlier class.

The following observation is the analog of Observation 3.19 in the context of interval orientations.

Observation 4.8 *Let Y_n, Y_1, and Y_c be as in the description of a three-way pivot. If $x \in Y_n$, then all edges of G_{1n} from x to $Y_1 \cup Y_c$ are neighbors of (z, x) in $I_\Delta(T)$, hence even-equivalent to (x, z). Similarly, if $y \in Y_1$, then all edges of G_{1n} from y to Y_c are even-equivalent to (y, z). In particular, if all edges of G_{1n} in $Y_n \cup Y_c \times \{z\}$ are even-equivalent, then so are all edges of G_{1n} that are directed from earlier to later sets in the sequence (Y_n, Y_1, Y_c).*

The following gives a similar insight about the effect of a modular pivot:

Lemma 4.9 *Let X_i, Y, x, and z be as in the definition of a modular pivot. All edges of G_{1n} that are directed from $X_i - Y$ to Y are in the same even-equivalence class as (x, z) in $I_\Delta(T)$.*

Proof. All members of $\{x\} \times Y$ are directed edges of G_n, and since z has edges of G_1 to all of $\{x\} \cup Y$, it is immediate that these edges are neighbors of (z, x) in $I_\Gamma(T)$, hence even-equivalent with (x, z).

Let ab be an arbitrary edge of G_{1n} such that $a \in X_i - Y$, $a \neq x$, and $b \in Y$. Note that ax is not an edge of G_n, by the definition of Y.

If ab is an edge of G_n, then ab, xb are both edges in G_n and ax is not an edge of G_n. Thus, (a, b) is even-equivalent to (x, b), and transitively, even-equivalent to (x, z).

Suppose ab is an edge of G_1. Then ax is either an edge in G_1 or in G_c, but in either of these cases, (a, b) and (b, x) are neighbors in $I_\Delta(T)$, hence (a, b) and (x, b) are even-equivalent. Transitively, (a, b) is even-equivalent to (x, z). Q.E.D.

The following is analogous to Lemma 3.21.

Lemma 4.10 *Let T be Delta-prime, and let $(v_1, v_2, \ldots, v_n)$ be an ordering of the vertices returned by the variant of Algorithm 3.20, that uses three-way and modular pivots in place of the standard pivot. Then the edges of G_{1n} that are directed from earlier to later vertices in this ordering are even-equivalent in $I_\Gamma(G)$.*

The proof differs only in trivial details from the proof of Lemma 3.21. The main difference is that it uses Observation 4.8 and Lemma 4.9 in place of Observation 3.19 at each place in the proof. The results of the section on comparability graphs are now easily adapted to the new context; the details of these proofs are also trivial and are left to the reader:

Theorem 4.11 *[23] Let T be an arbitrary intersection matrix.*

1. *If X and Y are two children of a degenerate node in the Delta tree that are adjacent in G_{1n}, then $(X \times Y) \cup (Y \times X)$ is a connected component of $I_\Delta(G)$. If T is an interval matrix, then $\{X \times Y, Y \times X\}$ is its bipartition.*
2. *If $X_1, X_2, \ldots, X_k$ are children of a prime node, then the set $E_Z = \{(a, b) : ab$ is an edge of G_{1n} and $x \in X_i, y \in X_j, i \neq j\}$ is a connected component of $I_\Delta(G)$. If T is an interval matrix, then there exists an ordering $X_{\pi(1)}, X_{\pi(2)}, \ldots, X_{\pi(k)}$ such that the elements of E_Z that go from an earlier child to a later child are one bipartition class and those that go from a later child to an earlier are another.*
3. *There are no other connected components of $I_\Delta(T)$.*

Corollary 4.12 *An intersection matrix is an interval matrix iff $I_\Delta(G)$ is bipartite.*

Lemma 4.13 *If T is an $n \times n$ intersection matrix that is not an interval matrix, then $I_\Delta(T)$ has an odd cycle of length at most $3n$.*

Recall that we can represent an interval realizer with a string, where each letter of the string is the name of a vertex whose interval has an endpoint there. Suppose T is an interval matrix. A strong Delta modules M of T has an analog in an interval realizer R of T: M is a set of intervals whose left endpoints are a consecutive substring and whose right endpoints are a consecutive substring. If M has a neighbor $y \in V - M$ in G_1, then interval y must have an endpoint inside all intervals in M and an endpoint outside all intervals in M. This constrains the intervals in M to have a common intersection, which forces $G_n|M$ to have no edges.

The weak modules are those that satisfy these requirements in some, but not all, realizers of T. The correspondence with Delta modules of T means that such sets define a quotient structure on the class of all interval realizers.

When one labels the nodes of the Delta tree with interval representations of their associated quotients, the tree gives a representation of an interval realizer (see Figure 15). This is obtained by a composition operation on the interval realizers themselves. This reflects the fact that the Delta modules define a quotient structure on interval realizers that mirror the quotient structure that they define on the intersection matrices of the interval realizers.

The quotient at a prime node can be reversed to obtain a representation of another quotient. A degenerate node induces a quotient that is complete in one of G_n, G_1, or G_c; it is easy to see how a realizer of such a graph can be permuted without changing such a simple represented quotient in T. Since each realizer of T is a string, the collection of all realizers of T is a language, and this tree gives a type of grammar for that language.

An algorithm is given in [22] for finding the modular decomposition of a graph that uses $I_\Gamma(G)$ to reduce the problem to the `Refine` operation of Algorithm 3.20. The similarities between the modular decomposition and its Γ relation, on the one hand, and the Delta tree and its Delta relation, on the other, can be used to reduce finding

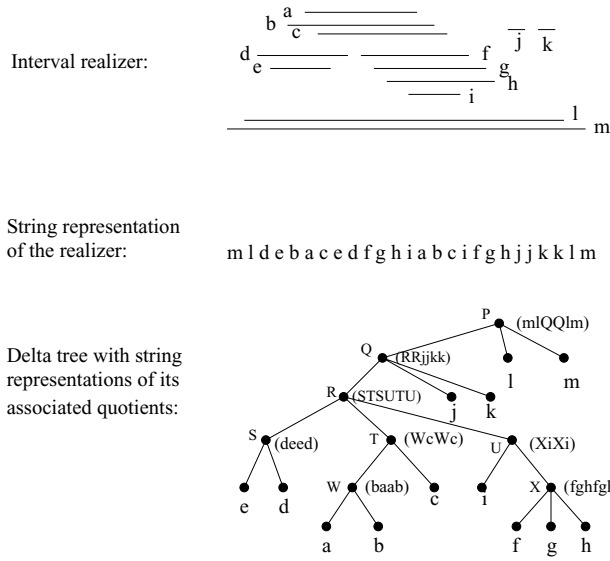


Figure 15. An interval realizer and the Delta tree for the intersection matrix given by a realizer R. When M is an internal node and $\mathcal{C}$ is its children, the node is labeled with the quotient $(R|M)/\mathcal{C}$, represented here with their string representations. By performing substitution operations in postorder, it is possible to reconstruct R in $O(n)$ time using a composition operation that mirrors the one defined by the quotient structure.

the Delta tree to the variant of `Refine` operation that uses the three-way and modular pivots in the same way. This, together with a linear-time algorithm for `Refine` that is given in [21] is the basis of an algorithm given in [17, 23] for finding the Delta tree in time linear in the size of the intersection graph G_{1c} that it represents. This assumes a sparse representation of the matrix consisting of labeling the edges of $G = G_{1c}$ with with 1's and c's.

5. Similar Forcing Relations in Other Structures

5.1. Zero-one Matrices and the Consecutive-ones Property

A zero-one matrix M has the **consecutive-ones property** if the columns of M can be permuted so that the 1s in each row are a consecutive block.

In this section, we summarize results from [18, 37] about an incompatibility relation on pairs of columns of an arbitrary zero-one matrix and a corresponding quotient structure on columns of the matrix. The relation and the quotient structure have a relationship analogous to that of Theorem 3.26: it is bipartite if and only if the matrix has the consecutive-ones property. In this case, the decomposition tree given by the quotient relation is the well-known **PQ tree**, which is a data structure for representing

the modules of a graph. However, the quotient structure is not limited to matrices that have the consecutive-ones property, so this gives a generalization of the PQ tree to arbitrary zero-one matrices.

Like the modular decomposition and its quotients, this generalized PQ tree, together with the quotients induced by children in their parent sets, gives a representation of the matrix, which can be recovered by taking a type of composition of the quotients [37].

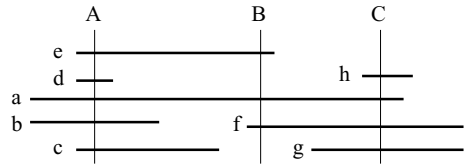
A graph is **chordal** if every simple cycle on four or more vertices has a **chord**, which is an edge not on the cycle that has its two endpoints on the cycle. Interval graphs are chordal.

If a graph is chordal, the sum of cardinalities of its maximal cliques is $O(m)$. Given a chordal graph, one may find the maximal cliques in linear time [32]. Since an interval graph is chordal, this applies to interval graphs also. Let the **clique matrix** be the matrix that has one row for each vertex of G, one columns for each maximal clique of G, and a 1 in row i and column j iff vertex i is a member of clique j.

Most of the algorithms that have appeared for recognizing interval graphs are based on the following reduction, which was described by Fulkerson and Gross:

Theorem 5.1 *[7] A graph is an interval graph iff its clique matrix has the consecutive-ones property.*

To see why Theorem 5.1 is true, note that the members of a clique correspond to a set of intervals that have a common point of intersection (see Figure 16). Thus, an interval realizer defines an ordering of cliques where clique C_i is before C_j in the ordering iff C_i's common intersection point precedes C_j's in the realizer. An interval x of a realizer will be a member of a clique C_i iff it contains C_i's intersection point. Since x is an interval, the cliques that contain x are consecutive in the ordering of cliques.



	A	B	C
a	1	1	1
b	1		
c	1		
d	1		
e	1	1	
f		1	1
g			1
h			1

Figure 16. Each maximal cliques in an interval graph correspond to a **clique point** on the line in an interval realizer. The **clique matrix** has a row for each vertex and a column for each maximal clique. The clique matrix can be found in linear time, given the graph. The order of the clique points gives a consecutive-ones ordering of the clique matrix. Conversely, a consecutive-ones ordering of the matrix gives an interval representation of the graph: the block of ones in each row gives one of the intervals in a realizer.

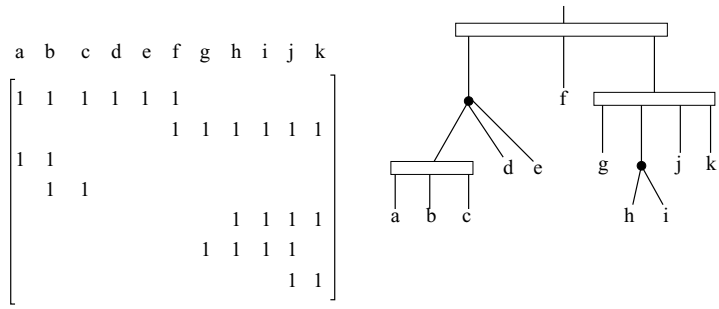


Figure 17. The PQ tree is a way of representing all consecutive-ones orderings of the columns of a matrix. The leaf order of the tree gives a consecutive-ones ordering. Permuting the order of children of a P node (black circles) gives a new order. For instance, by permuting the order of the left child of the root, we see that $(d, a, b, c, e, f, g, h, i, j, k)$ is a consecutive-ones ordering. Reversing the order of children of a Q node (horizontal bars) gives a new consecutive-ones order. For instance, by reversing the order of the right child of the root, we see that $(a, b, c, d, e, f, k, j, h, i, g)$ is a consecutive-ones ordering. The consecutive-ones orderings of the matrix are precisely the set of leaf orders obtainable by a sequence of such operations on the PQ tree.

With these observations, it is easy to see how to construct an interval realizer for G, given a consecutive-ones ordering of its clique matrix.

A consecutive-ones ordering of a matrix is a certificate that the matrix has the consecutive-ones property. A certificate that a matrix does not have the consecutive-ones property was given by Tucker in 1972 [33], but none of the algorithms that have appeared for recognizing consecutive-ones matrices produces a certificate of rejection.

Most algorithms for finding a consecutive-ones ordering of a matrix are based on construction of the PQ tree, which is illustrated in Figure 17). The leaves represent the columns of the matrix, and the internal nodes of the tree of one of two types: P-nodes and Q-nodes. The children of each internal node are ordered from left to right. The left-to-right order of leaves in the tree represents a consecutive-ones ordering of the matrix. To obtain other consecutive-ones orderings of the matrix, one can reverse the left-to-right order of children at any set of Q nodes, and permute arbitrarily the left-to-right order of children of any set of P nodes. This imposes a new left-to-right order on the leaves, which will be a new consecutive-ones ordering of the matrix.

The PQ tree is constructed in a way that ensures that all orderings of leaves obtainable in this way, and only such orderings, are consecutive-ones orderings of the matrix.

If M has duplicates of a row, we may delete all but one copy, as they place no additional restrictions on the problem. Henceforth, let us assume that M has no duplicate rows.

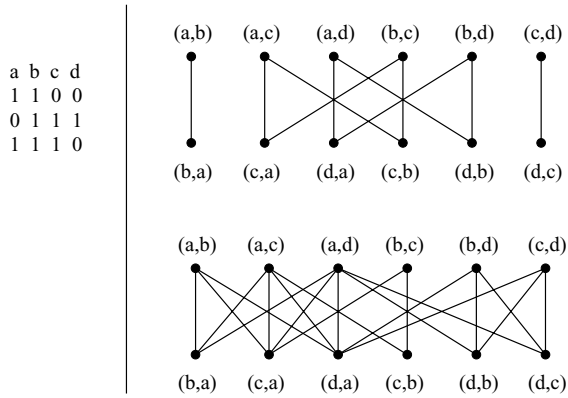


Figure 18. The incompatibility graph $I_C(M)$ of a 0-1 matrix has one vertex for each ordered pair of columns. Each ordered pair (x, y) is incompatible with (adjacent to) its twin (y, x), since at most one of these can reflect their relative order in a consecutive-ones ordering. In addition, if, in some row, x is 1, y is 0, and z is 1, then (x, y) and (y, z) are incompatible, since the order (x, y, z) would place a zero between two ones in that row. The first graph shows the incompatibility graph for the first row of the matrix, while the second shows the incompatibility graph for the whole matrix.

Definition 5.2 *[18] The* **incompatibility graph** $I_C(M)$ *of a 0-1 matrix M with n columns is the undirected graph that is defined as follows: (See Figure 18.)*

- *The vertex set of $I_C(G)$ is the set $\{(a, b)|a, b$ are distinct columns of $M\}$.*
- *The edge set of $I_C(G)$ are pairs of vertices of the following forms:*
  - *$\{\{(a, b), (b, a)\}|a$ and b are columns of $M\}$;*
  - *$\{\{(a, b), (b, c)\}|$ a, b, and c are columns of M and there exists a row that has 1's in columns a and c and a zero in column $b\}$.*

In a consecutive-ones ordering of columns of a matrix, the pairs of columns of the form $\{(i, j)|i$ is to the left of j in the ordering$\}$ must be an independent set in $I_C(M)$. Otherwise, there exists three columns $i, j, k : i < j < k$ and some row r where a zero in column j occurs in between 1's in columns i and k. Since reversing the order of columns in a consecutive-ones ordering produces another consecutive-ones ordering, it is obvious that $I_C(M)$ must be bipartite if a matrix M has the consecutive-ones property. As with the analogous result for the Gamma- and Delta-incompatibility relation, what is not obvious is that this condition is also sufficient.

Definition 5.3 *Let $V(M)$ denote the set of columns of a 0-1 matrix M. Let the* **row set** *associated with a row of M be the set of columns where a 1 occurs in the row. Let $\mathcal{R}(M)$ denote the family of row sets of M.*

A set $X \subseteq V(M)$ is a **column module** *if it does not overlap with any row set of M. Let $\mathcal{C}(M)$ denote the family of column modules of M.*

Theorem 5.4 *[37] The column modules of a consecutive-ones matrix M are a rooted set family on $V(M)$, and its decomposition tree is the PQ tree for M, where the degenerate nodes are the P nodes and the prime nodes are the Q nodes.*

However, it is just as easy to show that when M is an arbitrary 0-1 matrix, its column modules are still a rooted set family. The corresponding decomposition tree therefore gives a generalization of the PQ tree to arbitrary 0-1 matrices, called the PQR tree [37].

Let us define a congruence partition of a 0-1 matrix to be a partition $\mathcal{P}$ of its columns where every partition class is a column module. To try to show an associated quotient structure, one must come up with a suitable definition of the quotient $M/\mathcal{P}$ induced in M by a congruence partition. Given the definitions of quotients from previous sections, the obvious approach is to select a set A of arbitrary representatives, one from each member of $\mathcal{P}$, and to let the quotient be the submatrix $M[A]$. However, this does not work, as $M[A]$ can depend on the exact choice of representatives.

For instance, in Figure 17, $\mathcal{P} = \{\{a, b, c, d, e\}, \{f\}, \{g, h, i, j, k\}\}$. However, $M[\{a, f, g\}]$ is not the same matrix as $M[\{b, f, j\}]$, for example.

Fortunately, if one also restricts the rows, a definition of a quotient with all the desired properties emerges. Since $\mathcal{P}$ is a congruence partition, no row overlaps any member of $\mathcal{P}$. Thus, each row set is either a union of more than one member of $\mathcal{P}$, or a subset of a single member of $\mathcal{P}$. Let R be the set of rows whose row sets are unions of more than one member of $\mathcal{P}$. The matrix $(M[R])[A]$ is independent of the set A of representatives chosen from $\mathcal{P}$. For $X \in \mathcal{P}$, let R_X denote the set of rows whose row sets are subsets of X. The **factor** associated with X is $(M[R_X])[X]$.

Clearly, M can be recovered by a simple composition operation on the the quotient and the factors, except for the order of rows in M. The row order can be recovered if the position of the corresponding row in M is given for each row of a quotient or factor.

Let X be a node of the PQR tree, and let $\mathcal{C}$ denote its children. As in the case of modular decomposition, we can label the X with the associated quotient $(M[X])/\mathcal{C}$. This is illustrated in Figure 19 for the PQ tree of Figure 17. This gives a representation of M, as M can be recovered from the tree and this labeling.

In order to define a quotient structure, we need a definition of a restriction operator that is consistent with our definition of a quotient.

Definition 5.5 *If M is a 0-1 matrix and X is a set of columns, then let R be the set of rows whose row set is a subset of X. Let $M|_C X$, denote the matrix $(M[R])[X]$.*

Given the foregoing definitions, the following is not difficult to show.

Theorem 5.6 *Let $\mathcal{M}$ be the class of all 0-1 matrices and $|_C$ be the quotient and restriction operators defined for 0-1 matrices above, let $V(M)$ denote the column set*

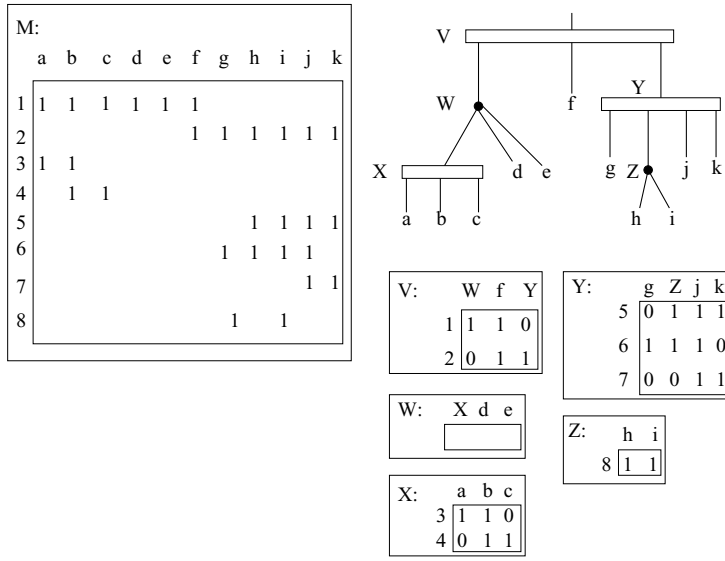


Figure 19. The PQR tree gives a decomposition of a 0-1 matrix into a series of quotient submatrices, one for each internal node of the tree. The quotient at the root is given by $(M[Y])[X]$, where X is the set of rows whose row set contains more than one child, and Y consists of one representative column from each child. For each child Z, the subtree rooted at Z is found by recursion on $(M[Z])[R_Z]$, where R_Z is the set of rows whose row sets are contained in Z. The degenerate nodes are those whose matrix is either a row of 1's (*e.g.* Z) or whose set of rows is empty (*e.g.* W). A matrix has the consecutive-ones property iff each of these quotients do.

of a 0-1 matrix M, and let $\mathcal{C}(M)$ denote its column modules. Then $(\mathcal{M}, V(), \mathcal{C}(), |_C)$ is a quotient structure.

Clearly, for any set family $\mathcal{F}$ on a universe V, a 0-1 matrix M can be constructed to represent the family as $\mathcal{R}(M)$. The column modules of a matrix are defined on an arbitrary set family. Thus, we can just as easily talk about the family $\mathcal{C}(\mathcal{F})$ of column modules of $\mathcal{F}$. This raises an interesting issue, which is that since $\mathcal{C}(\mathcal{F})$ is itself a family of sets, $\mathcal{C}(\mathcal{C}(\mathcal{F}))$ is well defined. What is its relationship to $\mathcal{C}(\mathcal{F})$?

It turns out that $\mathcal{C}(\mathcal{C}(\mathcal{F}))$ is the closure of $\mathcal{F}$ under the operations of adding the union, intersection, and symmetric difference of overlapping members of $\mathcal{F}$. In other words, $\mathcal{C}(\mathcal{C}(\mathcal{F}))$ is the minimal rooted set family that has $\mathcal{F}$ as a subfamily.

One consequence of this is that if $\mathcal{F}$ is itself a rooted set family, $\mathcal{C}(\mathcal{C}(\mathcal{F})) = \mathcal{F}$, and we may consider $\mathcal{F}$ and $\mathcal{C}(\mathcal{F})$ to be **dual** rooted set families. The decomposition tree of $\mathcal{C}(\mathcal{F})$ can be obtained from the decomposition tree of $\mathcal{F}$ by exchanging the roles of degenerate and prime labels on the nodes.

Using the techniques of the previous sections, applied to this new quotient structure and its forcing relation, we obtain analogs of the main results:

Theorem 5.7 *[18] Let M be an arbitrary 0-1 matrix and $I_C(M)$ be is its incompatibility graph, and let $\mathcal{T}$ be its PQR tree. Then:*

1. *If X and Y are two children of a degenerate node, then $(X \times Y) \cup (Y \times X)$ is a connected component of $I_C(M)$;*
2. *If $X_1, X_2, \ldots, X_k$ are the children of a prime node Z, then the set $\{(a, b) : x \in X_i, b \in X_j, i \neq j\}$ is a connected component of $I_C(M)$;*
3. *There are no other connected components of $I_C(M)$.*

Corollary 5.8 *A 0-1 matrix M has the consecutive-ones property if and only if $I_C(M)$ is bipartite.*

Apart from its theoretical interest, the corollary defines a certificate when M is not a consecutive-ones matrix. A different certificate, based on **asteroidal triples**, was given previously in [33]. However, to our knowledge, no algorithm has been published to produce it. Let m denote the number of rows, n denote the number of columns, and e denote the number of 1's in M. An $O(n + m + e)$ algorithm to compute the generalized PQ tree for an arbitrary 0-1 matrix, a certificate in the form of a consecutive-ones ordering when M has the property, and a certificate in the form of an odd cycle of length at most $3n$ in $I_C(M)$ if it is not is given in [18]. A key ingredient is Algorithm 3.20, suitably modified with a pivot operation that is appropriate to the new domain.

5.2. Probe Interval Graphs

Recently, a variation of the problem of finding an interval realizer for an interval graph has arisen from a biological application. Let V be a collection of fragments of DNA. A subset P of the fragments are designated as **probes** and the remaining set N of fragments are designated as **non-probes**. For each probe fragment, the other fragments of $P \cup N$ that the probe intersects on the genome are known, while the intersections between pairs of non-probes are unknown. Using these data, a biologist would like to determine the linear arrangement, or *physical mapping*, of the fragments in the genome that they came from.

The data for this problem can be represented with **probe interval graph**. A graph G is a probe interval graph with respect to a partition $\{P, N\}$ of the vertices if there exist a set of intervals, one for each vertex, such that for $x, y \in P \cup N$, xy is an edge of G iff the interval for x intersects the interval for y and *at least one of x and y is a member of P* (see Figure 20). If G is a probe-interval graph with respect to the partition, then a set of intervals is a a **probe-interval realizer** of G.

This application motivates the problem of finding a set of intervals that realizes the graph. If all fragments are probes, this problem is just that of finding an interval realizer of an interval graph. However, for a variety of technical and economic reasons, there is an advantage to being able to find a realizer when only a subset of the fragments are probes.

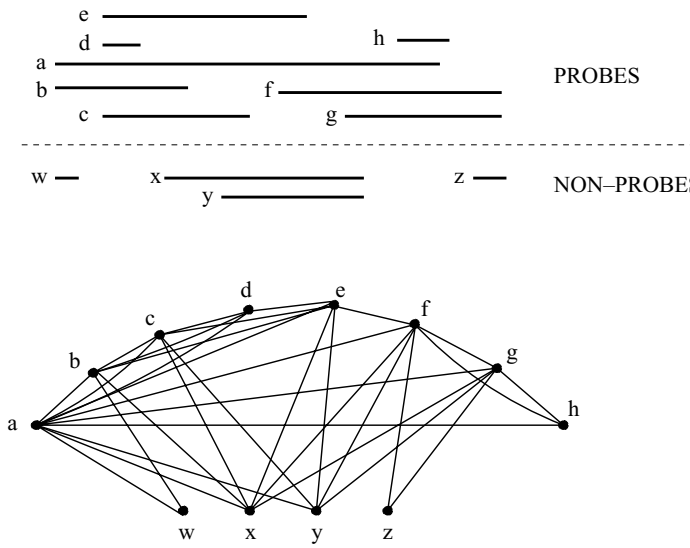


Figure 20. A **probe interval graph** is a graph with two sets of vertices: probes and non-probes. The edges can be represented with a set of intervals, one for each vertex, such that two vertices are adjacent if their corresponding intervals intersect **and at least on of the two vertices is a probe.** In this example, x and y are not adjacent, even though their corresponding intervals intersect.

Another application of for finding an interval representation of probe-interval graphs is recognition of circular-arc graphs. The solution outlined below is a key step in finding an intersection matrix of a circular-arc graph, which is the first step in the algorithm for recognizing circular-arc graphs given in [17, 23].

A polynomial time bound for finding a representation with intervals when P and N are given was first shown with an $O(n^2)$ algorithm in [13]. A subsequent $O(n + m \log n)$ bound is given in [25]. Note that in the biological application, the partition $\{P, N\}$ is given. Little is known about the complexity of the problem when P and N are not specified, and we do not deal with this problem here. Abusing the terminology slightly, we will call this a graph a probe interval graph if it has a probe interval realizer *with respect to the given partition* $\{P, N\}$.

The key insight for the $O(n + m \log n)$ bound of [25] is the following. Given only the probe intervals in a probe-interval realizer it is trivial to insert non-probe intervals to obtain a full probe-interval realizer: if the neighbors of a non-probe z is a clique, they have a common intersection point, and a small non-probe interval for z can be placed there; otherwise the interval for z is extended from the leftmost right endpoint to the rightmost left endpoint among the neighbors of z.

The probe intervals are an interval realizer of the interval graph $G|P$. One strategy for finding a probe-interval realizer from scratch might therefore be to find an interval realizer of $G|P$ and then to use the foregoing observation to extend it by adding non-probe

intervals. The problem with this approach is that not all interval realizers of $G|P$ can be extended to probe-interval realizers in this way. If z is a non-probe, an interval realizer of $G|P$ that places a non-neighbor of z between two neighbors cannot be extended, as z's adjacencies with P cannot be represented by inserting a single interval for z.

Let us call an interval realizer of $G|P$ **extensible** if it can be extended to a probe-interval realizer of G by adding intervals for the non-probes. Since it is easy to do this in this case, the problem of finding a probe-interval realizer of G reduces to that of finding an extensible realizer of $G|P$.

Let T denote the intersection matrix of the probe-interval realizer when the distinction between intervals in P and N is ignored. Let the graphs G_n, G_1, and G_c be defined by this matrix as described in Section 4. Let Q denote the set of entries of T that give the relationship between pairs of intervals that contain at least one probe, and let H_n, H_1, and H_c denote the result of removing edges of G_n, G_1, and G_c that are not incident to a probe vertex. Let us call Q a **probe intersection matrix**.

Clearly, any interval realizer of T is a probe-interval realizer of G. If T were known, we could therefore use the algorithm of Section 4 to find a probe-interval realizer. In [25], an $O(n + m \log n)$ algorithm is given for finding Q. Though this is only part of the information in T, for each non-probe z, Q completely specifies $T|(P \cup \{z\})$, and this is enough to allow us to derive an extensible realizer of $G|P$.

An interval realizer of T gives an interval orientation to edges of G_{1n}. Let a **probe orientation** be the restriction of this orientation to edges of H_{1n}. Its further restriction to edges of $G_{1n}|P$ is an interval orientation of $G_{1n}|P$, and the corresponding interval realizer of $T|P$ must be an extensible realizer. Let us call this an **extensible orientation** of $G_{1n}|P$. It therefore suffices to find a probe orientation of H_{1n} in order to find an extensible realizer, which allows us to find a probe interval realizer of G. The following gives the required constraints:

Definition 5.9 *The probe-incompatibility graph of T is the undirected graph $I_P(T)$ that is defined as follows:*

- *The vertex set of $I_P(T)$ is the set of edges of H_{1n}.*
- *Two vertices of $I_P(T)$ are adjacent if and only if they are adjacent in $I_\Delta(T|(P + z))$ for some $z \in N$.*

Lemma 5.10 *A probe orientation of H_{1n} must be independent in $I_P(T)$.*

Proof. A probe-interval realizer of T gives an interval orientation of G_{1n}, which is independent in $I_\Delta(T)$. $I_P(T)$ is a restriction of this to edges of H_{1n}. Q.E.D.

Using techniques similar to those of Section 4, it can be shown that this condition is sufficient: any acyclic orientation of the edges of H_{1n} is a probe orientation.

To represent all probe orientations, we may define a quotient structure that gives a compact representation of the components of I_P, as well as their bipartitions if T has a probe interval realizer:

Definition 5.11 *Q be a probe intersection matrix with probe set $P(T)$. A set $X \subseteq P$ is an* **extensible module** *in Q if for every $z \in N$ either X or $X + z$ is a Delta module in $T|_T(P + z)$. For $X \subseteq P$, let $Q|_P X$ denote $Q|(X \cup N)$, and let $\mathcal{F}(Q)$ denote the extensible modules of Q.*

Theorem 5.12 *Let $\mathcal{Q}$ be the family of probe intersection matrices. $(\mathcal{Q}, P(), \mathcal{F}(), |_P)$ is a quotient structure.*

Note that the decomposition tree implied by Theorem 5.12 has one leaf for each member of P, not for each member of $P \cup N$. It has the same role with respect to extensible orientations of $G_{1n}|P$ as the Delta tree has with respect to interval orientations of G_{1n}. When the children of prime nodes are ordered appropriately, the tree gives a similar representation of all extensible orientations of $G_{1n}|P$, and for each of these, an extensible realizer can be obtained by the techniques of Section 4. Like the Delta tree, this tree can be computed with an adaptation of a modular decomposition algorithm that is general to quotient structures.

If G_{1n} does not have a probe orientation, then $I_P(T)$ has an odd cycle of length at most $3n$. However, this is not a certificate that G is not a probe interval graph, since the odd cycle could also result from an erroneous computation of the probe interval matrix. We do not know of a compact certificate for showing that the entries of the probe intersection matrix are correct.

References

[1] S. Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. U.S.A.*, 45: 1607–1620 (1959).

[2] K.S. Booth. *PQ-Tree Algorithms*. PhD thesis, Department of Computer Science, U.C. Berkeley, (1975).

[3] S. Booth and S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13: 335–379 (1976).

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, Boston, (2001).

[5] B. Duschnik and E. W. Miller. Partially ordered sets. *Amer. J. Math.*, 63: 600–610 (1941).

[6] E.M. Eschen and J.P. Spinrad. An $O(n^2)$ algorithm for circular-arc graph recognition. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 4: 128–137 (1993).

[7] D.R. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15: 835–855 (1965).

[8] T. Gallai. Transitiv orientierbare Graphen. *Acta Math. Acad. Sci. Hungar.*, 18: 25–66 (1967).

[9] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, (1980). Second edition: *Annuals of Discrete Mathematics, 57, Elsevier, Amsterdam* (2004).

[10] M. Habib, R.M. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234: 59–84 (2000).

[11] W.L. Hsu. $O(mn)$ algorithms for the recognition and isomorphism problems on circular-arc graphs. *SIAM J. Comput.*, 24: 411–439 (1995).

[12] W.L. Hsu and R.M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296: 59–74 (2003).

[13] J.L. Johnson and J.P. Spinrad. A polynomial time recognition algorithm for probe interval graphs. *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, 12: 477–486 (2001).

[14] D. Kratsch, R.M. McConnell, K. Mehlhorn, and J.P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, (2003) pp. 866–875.

[15] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamata Mathematicae*, (1930) pp. 271–283.

[16] E. Marczewski. Sur deux proprietés des classes d'ensembles. *Fund. Math.*, 33: 303–307 (1945).

[17] R. M. McConnell. Linear-time recognition of circular-arc graphs. *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS01)*, 42: 386–394 (2001).

[18] R. M. McConnell. A certifying algorithm for the consecutive-ones property. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA04)*, 15: 761–770 (2004).

[19] R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5: 536–545 (1994).

[20] R. M. McConnell and J. P. Spinrad. Linear-time transitive orientation. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 8: 19–25 (1997).

[21] R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3): 189–241 (1999).

[22] R. M. McConnell and J. P. Spinrad. Ordered vertex partitioning. *Discrete Math and Theoretical and Computer Science*, 4: 45–60 (2000).

[23] R.M. McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37: 93–147 (2003).

[24] R.M. McConnell and F de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Math, to appear*.

[25] R.M. McConnell and J.P. Spinrad. Construction of probe interval models. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, (2002) pp. 866–875.

[26] K. Mehlhorn. The theory behind LEDA. keynote address. *Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 13: 866–875 (2002).

[27] K. Mehlhorn and S. Näeher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, (1999).

[28] R. H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Order*, pp. 41–101. D. Reidel, Boston, (1985).

[29] R. H. Möhring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operations Research*, 4: 195–225 (1985).

[30] R. H. Möhring and F. J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19: 257–356 (1984).

[31] A. Pnueli, A. Lempel, and S. Even. Transitive orientation of graphs and identification of permutation graphs. *Canad. J. Math.*, 23: 160–175 (1971).

[32] D. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5: 266–283 (1976).

[33] A. Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory, Series B*, 12: 153–162 (1972).

[34] A. Tucker. An efficient test for circular-arc graphs. *SIAM Journal on Computing*, 9: 1–24 (1980).

[35] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44: 826–849 (1997).

[36] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM J. Algebraic and Discrete Methods*, 3: 303–322 (1982).

[37] J. Meidanis, O. Porto, G. P. Telles. On the consecutive ones property. *Discrete Applied Math*, 88: 325–354 (1997).

5

The *Local Ratio* Technique and Its Application to Scheduling and Resource Allocation Problems

Reuven Bar-Yehuda
Keren Bendel
Department of Computer Science
Technion, Haifa 32000, Israel

Ari Freund
IBM Haifa Research Lab,
Haifa University Campus
Haifa 31905, Israel

Dror Rawitz
School of Electrical Engineering
Tel-Aviv University
Tel-Aviv 69978, Israel

Abstract

We give a short survey of the local ratio technique for approximation algorithms, focusing mainly on scheduling and resource allocation problems.

1. Introduction

The *local ratio* technique is used in the design and analysis of approximation algorithms for NP-hard optimization problems. Since its first appearance in the early 1980's it has been used extensively, and recently, has been fortunate to have a great measure of success in dealing with scheduling problems. Being simple and elegant, it is easy to understand, yet has surprisingly broad applicability.

At the focus of this chapter are applications of the *local ratio* technique to scheduling problems, but we also give an introduction to the technique and touch upon some of the milestones in its path of development. We present a host of optimization problems and local-ratio approximation algorithms, some of which were developed using the technique, and others which are local ratio interpretations of pre-existing algorithms.

The basic concepts relating to optimization and approximation are as follows. (Precise definitions can be found in Section 2.) An *optimization problem* is a problem in which every possible solution is associated with a cost, and we seek a solution of minimum (or maximum) cost. For example, in the *minimum spanning tree* problem our objective is to find a minimum cost spanning tree in a given edge weighted graph. For this problem, the solutions are all spanning trees, and the cost of each spanning tree is its total edge weight. Although this particular problem is polynomial-time solvable, many other optimization problems are NP-hard, and for those, computing *approximate* solutions (efficiently) is of interest. (In fact, finding approximate solutions is also of interest in cases where this can be done faster than finding exact solutions.) A solution whose cost is within a factor of r of the optimum is said to be *r-approximate*. Thus, for example, a spanning tree whose weight is at most twice the weight of a minimum spanning tree is said to be 2-approximate. An *r-approximation* algorithm is one that is guaranteed to return r-approximate solutions.

Analyzing an r-approximation algorithm consists mainly in showing that it achieves the desired degree of approximation, namely, r (correctness and efficiency are usually straightforward). To do so we need to obtain a lower bound B on the optimum cost and show that cost of the solution is no more than $r \cdot B$. The local ratio technique uses a "local" variation of this idea as a design principle. In essence, the idea is to break down the cost W of the algorithm's solution into a sum of "partial costs" $W = \sum_{i=1}^{k} W_i$, and similarly break down the lower bound B into $B = \sum_{i=1}^{k} B_i$, and to show that $W_i \leq r \cdot B_i$ for all i. (In the maximization case, B is an upper bound and we need to show that $W_i \geq B_i/r$ for all i.) The breakdown of W and B is determined by the way in which the solution is constructed by the algorithm. In fact, the algorithm constructs the solution in such a manner as to ensure that such breakdowns exist. Put differently, at the ith step, the algorithm "pays" $W_i \leq r \cdot B_i$ and manipulates the problem instance so that the optimum cost drops by at least B_i.

To see how this works in practice we demonstrate the technique on the *vertex cover* problem. Given a graph $G = (V, E)$, a *vertex cover* is a set of vertices $C \subseteq V$ such that every edge has at least one endpoint in C. In the vertex cover problem we are given a graph $G = (V, E)$ and a non-negative cost function w on its vertices, and our goal is to find a vertex cover of minimum cost. Imagine that we have to actually purchase the vertices we select as our solution. Rather than somehow deciding on which vertices to buy and then paying for them, we adopt the following strategy. We repeatedly select a vertex and pay for it. However, the amount we pay need not cover its entire cost; we may return to the same vertex later and pay some more. In order to keep track of the payments, whenever we pay ϵ for a vertex, we lower its marked price by ϵ. When the marked price of a vertex drops to zero, we are free to take it, as it has been fully paid for.

The heart of the matter is the rule by which we select the vertex and decide on the amount to pay for it. Actually, we select two vertices each time and pay ϵ for each, in the following manner. We select any edge (u, v) whose two endpoints have non-zero cost, and pay $\epsilon = \min\{\text{price of } u, \text{ price of } v\}$ for both u and v. As a result, the marked price of at least one of the endpoints drops to zero. After $O(|V|)$ rounds, prices drop sufficiently so that every edge has an endpoint of zero cost. Hence, the set of all zero-cost vertices is a vertex cover. We take this zero-cost set as our solution.

We formalize this by the following algorithm. We say that an edge is *positive* if both its endpoints have strictly positive cost.

Algorithm VC

1. While there exists a positive edge (u, v):
2. Let $\epsilon = \min\{w(u), w(v)\}$.
3. $w(u) \leftarrow w(u) - \epsilon$.
4. $w(v) \leftarrow w(v) - \epsilon$.
5. Return the set $C = \{v | w(v) = 0\}$.

To analyze the algorithm, consider the ith iteration. Let (u_i, v_i) be the edge selected in this iteration, and let ϵ_i be the payment made for each endpoint. Every vertex cover must contain either u_i or v_i (in order to cover the edge (u_i, v_i)), and therefore decreasing the price of these vertices by ϵ_i lowers the cost of every vertex cover by at least ϵ_i. It follows that the optimum, denoted *OPT*, also decreases by at least ϵ_i. Thus, in the ith round we pay $2\epsilon_i$ and lower *OPT* by at least ϵ_i, so the *local ratio* between our payment and the drop in *OPT* is at most 2 in any given iteration. Summing over all iterations, we get that the ratio between our total payment and the total drop in *OPT* is at most 2 as well. Now, we know that the total drop in the optimal cost is *OPT*, since we end up with a vertex cover of zero cost, so our total payment is at most $2OPT$. Since this payment fully covers the solution's cost (in terms of the original cost function), the solution is 2-approximate.

It is interesting to note that the proof that the solution found is 2-approximate does not depend on the actual value of ϵ in any given iteration. In fact, any value between 0 and $\min\{w(u), w(v)\}$ would yield the same result (by the same arguments). We chose $\min\{w(u), w(v)\}$ for the sake of efficiency. This choice ensures that the number of vertices with positive cost strictly decreases with each iteration.

We also observe that the analysis of algorithm **VC** does not seem tight. The analysis bounds the cost of the solution by the sum of *all* payments, but some of these payments are made for vertices that do not end up in the solution. It might seem that trying to "recover" these wasteful payments could yield a better approximation ratio, but this is not true (in the worst case); it is easy to construct examples in which all vertices for which payments are made are eventually made part of the solution.

Finally, there might still seem to be some slack in the analysis, for in the final step of the algorithm *all* zero-cost vertices are taken to the solution, without trying to remove

unnecessary ones. One simple idea is to prune the solution and turn it into a *minimal* (with respect to set inclusion) subset of C that is still a vertex cover. Unfortunately, it is not difficult to come up with worst-case scenarios in which C is minimal to begin with. Nevertheless, we shall see that such ideas are sometimes useful (and, in fact, necessary) in the context of other optimization problems.

1.1. Historical Highlights

The origins of the local ratio technique can be traced back to a paper by Bar-Yehuda and Even on *vertex cover* and *set cover* [16]. In this paper, the authors presented a linear time approximation algorithm for *set cover*, that generalizes Algorithm **VC**, and presented a primal-dual analysis of it. This algorithm was motivated by a previous algorithm of Hochbaum [42], which was based on LP duality, and required the solution of a linear program. Even though Bar-Yehuda and Even's primal-dual analysis contains an implicit local ratio argument, the debut of the local ratio technique occurred in a followup paper [17], where the authors gave a local ratio analysis of the same algorithm. They also designed a specialized $(2 - \frac{\log_2 \log_2 n}{2 \log_2 n})$-approximation algorithm for *vertex cover* that contains a local-ratio phase. The technique was dormant until Bafna, Berman, and Fujito [9] incorporated the idea of *minimal solutions* into the local ratio technique in order to devised a local ratio 2-approximation algorithm for the *feedback vertex set* problem. Subsequently, two generic algorithms were presented. Fujito [33] gave a unified local ratio approximation algorithm for node-deletion problems, and Bar-Yehuda [15] developed a local ratio framework that explained most local ratio (and primal-dual) approximation algorithms known at the time. At this point in time the local ratio technique had reached a certain level of maturity, but only in the context of minimization problems. No local ratio algorithms were known for maximization problems. This was changed by Bar-Noy et al. [11], who presented the first local ratio (and primal-dual) algorithms for maximization problems. These algorithm are based on the notion of *maximal* solutions rather than minimal ones. More recently, Bar-Yehuda and Rawitz [19] developed two approximation frameworks, one extending the generic local ratio algorithm from [15], and the other extending known primal-dual frameworks [38, 22], and proved that both frameworks are equivalent, thus merging these two seemingly independent lines of research. The most recent local ratio development, due to Bar-Yehuda et al. [12], is a novel extension of the local ratio technique called *fractional local ratio*.

1.2. Organization

The remainder of this chapter is organized as follows. In Section 2 we establish some terminology and notation. In Section 3 we state and prove the Local Ratio Theorem (for minimization problems) and formulate the local ratio technique as a design and analysis framework based on it. In Section 4 we formally introduce the idea of minimal solutions into the framework, making it powerful enough to encompass many known approximation algorithms for covering problems. Finally, in Section 5 we discuss local ratio algorithms for scheduling problems, focusing mainly on maximization problems. As a first step, we describe a local ratio framework for maximization problems, which

is, in a sense, a mirror image of its minimization counterpart developed in Sections 3 and 4. We then survey a host of problems to which the local ratio technique has been successfully applied.

In order not to interrupt the flow of text we have removed nearly all citations and references from the running text, and instead have included at the end of each section a subsection titled *Background*, in which we cite sources for the material covered in the section and discuss related work.

1.3. Background

The *vertex cover* problem is known to be NP-hard even for planar cubic graphs with unit weights [36]. Håstad [41] shows, using PCP arguments[1], that *vertex cover* cannot be approximated within a factor of $\frac{7}{6}$ unless P = NP. Dinur and Safra [29] improve this bound to $10\sqrt{5} - 21 \approx 1.36067$. The first 2-approximation algorithm for weighted *vertex cover* is due to Nemhauser and Trotter [57]. Hochbaum [43] uses this algorithm to obtain an approximation algorithm with performance ratio $2 - \frac{2}{d_{\max}}$, where $d_{\max}$ is the maximum degree of a vertex. Gavril (see [35]) gives a linear time 2-approximation algorithm for the non-weighted case. (Algorithm **VC** reduces to this algorithm on non-weighted instances.) Hochbaum [42] presents two 2-approximation algorithms, both requiring the solution of a linear program. The first constructs a vertex cover based on the optimal dual solution, whereas the second is a simple LP rounding algorithm. Bar-Yehuda and Even [16] present an LP based approximation algorithm for weighted *set cover* that does not solve a linear program directly. Instead, it constructs simultaneously a primal integral solution and a dual feasible solution without solving either the primal or dual programs. It is the first algorithm to operate in this method, a method which later became known as the *primal-dual schema*. Their algorithm reduces to Algorithm **VC** on instances that are graphs. In a subsequent paper, Bar-Yehuda and Even [17] provide an alternative local ratio analysis for this algorithm, making it the first local ratio algorithm as well. They also present a specialized $(2 - \frac{\log_2 \log_2 n}{2 \log_2 n})$-approximation algorithm for *vertex cover*. Independently, Monien and Speckenmeyer [56] achieved the same ratio for the unweighted case. Halperin [40] improved this result to $2 - (1 - o(1))\frac{2 \ln \ln d_{\max}}{\ln d_{\max}}$ using semidefinite programming.

2. Definitions and Notation

An optimization problem is comprised of a family of problem *instances*. Each instance is associated with (1) a collection of *solutions*, each of which is either *feasible* or *infeasible*, and (2) a *cost* function assigning a cost to each solution. We note that in the sequel we use the terms *cost* and *weight* interchangeably. Each optimization

[1] By "PCP arguments" we mean arguments based on the celebrated PCP Theorem and its proof. The PCP theorem [6, 5] and its variants state that certain suitably defined complexity classes are in fact equal to NP. A rather surprising consequence of this is a technique for proving lower bounds on the approximation ratio achievable (in polynomial time) for various problems. For more details see Arora and Lund [4] and Ausiello et al. [7].

problem can be either a *minimization* problem or a *maximization* problem. For a given problem instance, a feasible solution is referred to as *optimal* if it is either minimal or maximal (depending, respectively, on whether the problem is one of minimization or maximization) among all feasible solutions. The cost of an optimal solution is called the *optimum value*, or simply, the *optimum*. For example, in the well known *minimum spanning tree* problem instances are edge-weighted graphs, solutions are subgraphs, feasible solutions are spanning trees, the cost of a given spanning tree is the total weight of its edges, and optimal solutions are spanning trees of minimum total edge weight.

Most of the problems we consider in this survey can be formulated as problems of selecting a subset (satisfying certain constraints) of a given set of objects. For example, in the *minimum spanning tree* problem we are required to select a subset of the edges that form a spanning tree. In such problems we consider the cost function to be defined on the objects, and extend it to subsets in the natural manner.

An approximation algorithm for an optimization problem takes an input instance and efficiently computes a feasible solution whose value is "close" to the optimum. The most popular measure of closeness is the *approximation ratio*. Recall that for $r \geq 1$, a feasible solution is called *r-approximate* if its cost is within a factor of r of the optimum. More formally, in the minimization case, a feasible solution X is said to be r-approximate if $w(X) \leq r \cdot w(X^*)$, where $w(X)$ is the cost of X, and X^* is an optimal solution. In the minimization case, X is said to be r-approximate if $w(X) \geq w(X^*)/r$. (Note that in both cases r is smaller, when X is closer to X^*.) An r-approximate solution is also referred to as an *r-approximation*. An algorithm that computes r-approximate solutions is said to achieve an *approximation factor* of r, and it is called an *r-approximation* algorithm. Also, r is said to be a *performance guarantee* for it. The *approximation ratio* of a given algorithm is $\inf\{\mathrm{r} \mid \mathrm{r}$ is a performance guarantee for the algorithm.$\}$ Nevertheless, the term *approximation ratio* is sometimes used instead of *performance guarantee*.

We assume the following conventions, except where specified otherwise. All weights are non-negative and denoted by w. We denote by $w(x)$ the weight of element x, and by $w(X)$ the total weight of set X, i.e., $w(X) = \sum_{x \in X} w(x)$. We denote the optimum value of the problem instance at hand by *OPT*. All graphs are simple and undirected. A graph is denoted $G = (V, E)$, where $n \triangleq |V|$, and $m \triangleq |E|$. The degree of vertex v is denoted by $\deg(v)$.

3. The Local Ratio Theorem

In Algorithm **VC** we have paid $2 \cdot \epsilon$ for lowering *OPT* by at least ϵ in each round. Other local ratio algorithms can be explained similarly—one pays in each round at most $r \cdot \epsilon$, for some r, while lowering *OPT* by at least ϵ. If the same r is used in all rounds, the solution computed is r-approximate. This idea works well for several problems. However, it is not hard to see that this idea works mainly because we make a down payment on several items, and we are able to argue that *OPT* must drop by a proportional

amount because every solution must involve some of these items. This localization of the payments is at the root of the simplicity and elegance of the analysis, but it is also the source of its weakness: how can we design algorithms for problems in which no single set of items is necessarily involved in every optimal solution? For example, consider the *feedback vertex set* problem, in which we are given a graph and a weight function on the vertices, and our goal is to remove a minimum weight set of vertices such that the remaining graph contains no cycles. Clearly, it is not always possible to find two vertices such that at least one of them is part of every optimal solution! The Local Ratio Theorem, which is given below, allows us to go beyond localized payments by focusing on the *changes* in the weight function, and treating these changes as weight functions in their own right. Indeed, this is essential in the local ratio 2-approximation algorithm for feedback vertex set that is given in the next section.

The Local Ratio Theorem is deceptively simple. It applies to optimization problems that can be formulated as follows.

> Given a *weight vector* $w \in \mathbb{R}^n$ and a set of *feasibility constraints* $\mathcal{F}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{F}$ that minimizes the inner product $w \cdot x$.

(This section discusses minimization problems. In Section 5 we deal with maximization problems.)

In this survey we mainly focus on optimization problems in which $x \in \{0, 1\}^n$. In this case the optimization problem consists of instances in which the input contains a set I of n weighted elements and a set of feasibility constraints on subsets of I. Feasible solutions are subsets of I satisfying the feasibility constraints. The cost of a feasible solution is the total weight of the elements it contains. Such a minimization problem is called a *covering* problem if any extension of a feasible solution to any possible instance is always feasible. The family of covering problems contains a broad range of optimization problems, such as *vertex cover*, *set cover*, and *feedback vertex set*.

Theorem 1 (Local Ratio—Minimization Problems) *Let $\mathcal{F}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathcal{F}$) that is r-approximate with respect to w_1 and with respect to w_2. Then, x is r-approximate with respect to w as well.*

Proof. Let x^*, x_1^*, and x_2^* be optimal solutions with respect to w, w_1, and w_2, respectively. Clearly, $w_1 \cdot x_1^* \leq w_1 \cdot x^*$ and $w_2 \cdot x_2^* \leq w_2 \cdot x^*$. Thus,

$$\begin{aligned} w \cdot x &= w_1 \cdot x + w_2 \cdot x \leq r(w_1 \cdot x_1^*) + r(w_2 \cdot x_2^*) \leq r(w_1 \cdot x^*) + r(w_2 \cdot x^*) \\ &= r(w \cdot x^*) \end{aligned}$$

and we are done. ∎

As we shall see, algorithms that are based on the Local Ratio Theorem are typically recursive and has the following general structure. If a zero-cost solution can be found,

return one. Otherwise, find a decomposition of w into two weight functions w_1 and $w_2 = w - w_1$, and solve the problem recursively on w_2. We demonstrate this on the *vertex cover* problem. (Recall that a positive edge is an edge whose two endpoints have non-zero cost.)

Algorithm RecursiveVC(G,w)

1. If all edges are non-positive, return the set $C = \{v \mid w(v) = 0\}$.
2. Let (u, v) be a positive edge, and let $\epsilon \triangleq \min\{w(u), w(v)\}$.
3. Define $w_1(x) = \begin{cases} \epsilon & x = u \text{ or } x = v, \\ 0 & \text{otherwise,} \end{cases}$ and define $w_2 = w - w_1$.
4. Return **RecursiveVC**(G,w_2).

Clearly, Algorithm **RecursiveVC** is merely a recursive version of Algorithm **VC**. However, the recursive formulation is amenable to analysis that is based on the Local Ratio Theorem. We show that the the algorithm computes 2-approximate solutions by induction on the number of recursive calls (which is clearly finite). In the recursion base, the algorithm returns a zero-weight vertex cover, which is optimal. For the inductive step, consider the solution C. By the inductive hypothesis C is 2-approximate with respect to w_2. We claim that C is also 2-approximate with respect to w_1. In fact, *every* feasible solution is 2-approximate with respect to w_1. Observe that the cost (with respect to w_1) of every vertex cover is at most 2ϵ, while the minimum cost of a vertex cover is at least ϵ. Thus, by the Local Ratio Theorem C is 2-approximate with respect to w as well.

We remark that algorithms that follow the general structure outlined above differ from one another only in the choice of w_1. (Actually, the way they search for a zero-cost solution is sometimes different.) It is not surprising that these algorithms also share most of their analyses. Specifically, the proof that a given algorithm is an r-approximation is by induction on the number of recursive calls. In the base case, the solution has zero cost, and hence it is optimal (and also r-approximate). In the inductive step, the solution returned by the recursive call is r-approximate with respect to w_2 by the inductive hypothesis. And, it is shown that every solution is r-approximate with respect to w_1. This makes the current solution r-approximate with respect to w due to the Local Ratio Theorem. Thus, different algorithms are different from one another only in the choice of the weight function w_1 in each recursive call, and in the proof that every feasible solution is r-approximate with respect to w_1. We formalizes this notion by the following definition.

Definition 1 *Given a set of constraints $\mathcal{F}$ on vectors in $\mathbb{R}^n$ and a number $r \geq 1$, a weight vector $w \in \mathbb{R}^n$ is said to be* fully r-effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all feasible solutions x.*

We conclude this section by demonstrating the above framework on the *set cover* problem. Since the analysis of algorithms in our framework boils down to proving that w_1 is r-effective, for an appropriately chosen r, we focus solely on w_1, and neglect to

mention the remaining details explicitly. We remark that our algorithm for *set cover* can be formulated just as easily in terms of localized payments; the true power of the Local Ratio Theorem will become apparent in Section 4.

3.1. Set Cover

In the *set cover* problem we are given a collection of non-empty sets $\mathcal{C} = \{S_1, \ldots, S_n\}$ and a weight function w on the sets. A *set cover* is a sub-collection of sets that *covers* all elements. In other words, the requirement is that the union of the sets in the set cover be equal to $U \triangleq \bigcup_{i=1}^{n} S_i$. The objective is to find a minimum-cost set cover.

Let $\deg(x)$ be the number of sets in $\mathcal{C}$ that contain x, i.e., $\deg(x) = |\{S \in \mathcal{C} \mid x \in S\}|$. Let $d_{\max} = \max_{x \in U} \deg(x)$. We present a fully $d_{\max}$-effective weight function w_1. Let x be an element that is not covered by any zero-weight set, and let $\epsilon = \min\{w(S) \mid x \in S\}$. Define:

$$w_1(S) = \begin{cases} \epsilon & x \in S, \\ 0 & \text{otherwise.} \end{cases}$$

w_1 is $d_{\max}$-effective since (1) the cost of every feasible solution is bounded by $\epsilon \cdot \deg(x) \leq \epsilon \cdot d_{\max}$, and (2) every set cover must cover x, and therefore must cost at least ϵ.

Note that vertex cover can be seen as a set cover problem in which the sets are the vertices and the elements are the edges (and therefore $d_{\max} = 2$). Indeed, Algorithm **RecursiveVC** is a special case of the algorithm that is implied by the discussion above.

3.2. Background

For the unweighted *set cover* problem, Johnson [46] and Lovász [52] show that the greedy algorithm is an $H_{s_{\max}}$-approximation algorithm, where H_n the nth harmonic number, i.e., $H_n = \sum_{i=1}^{n} \frac{1}{i}$, and $s_{\max}$ is the maximum size of a set. This result was generalize by Chvátal [28] result to the weighted case. Hochbaum [42] gives two $d_{\max}$-approximation algorithms, both of which are based on solving a linear program. Bar-Yehuda and Even [16] suggest a linear time primal-dual $d_{\max}$-approximation algorithm. In subsequent work [17], they present the Local Ratio Theorem and provide a local ratio analysis of the same algorithm. (Their analysis is the one given in this section.) Feige [32] proves a lower bound of $(1 - o(1)) \ln |U|$ (unless NP$\subseteq$DTIME($n^{O(\log \log n)}$)). Raz and Safra [59] show that set cover cannot be approximated within a factor of $c \log n$ for some $c > 0$ unless P = NP.

4. A Framework for Covering Problems

In the problems we have seen this far, we were always able to identify a small subset of items (vertices or sets) and argue that every feasible solution must include at

least one of them. We defined a weight function w_1 that associated a weight of ϵ with each of the items in this small subset and a weight of zero with all others. Thus, we were able to obtain an approximation ratio bounded by the size of the subset. There are many problems, though, where it is impossible to identify such a small subset, since no such subset necessarily exists.

An good example is the *partial set cover* problem (or *partial cover* problem, for short). This problem is similar to *set cover* except that not all elements should be covered. More specifically, the input consists of a collection of sets, a weight function on the sets, and a number k, and we want to find a minimum-cost collection of sets that covers at least k of the elements. The crucial difference between *set cover* and *partial set cover* is that in the latter, there is no single element that must be covered by all feasible solutions. Recall that the algorithm for *set cover* picked some element x and defined a weight function w_1 that associated a weight of ϵ with each of the sets that contains x. The analysis was based on the fact that, with respect to w_1, ϵ is a lower bound, since x must be covered, while $\epsilon \cdot \deg(x)$ is an upper bound on the cost of every set cover. This approach fails for *partial set cover*—an optimal solution need not necessarily cover x, and therefore ϵ is no longer a lower bound. Thus if we use w_1, we will end up with a solution whose weight is positive, while *OPT* (with respect to w_1) may be equal to 0.

We cope with such hard situations by extending the same upper-bound/lower-bound idea. Even if we cannot identify a small subset of items that must contribute to all solutions, we know that the set of *all* items must surely do so (since, otherwise, the empty set is an optimal solution). Thus, to prevent *OPT* from being equal to 0, we can assign a positive weight to every item (or at least to many items). This takes care of the lower bound, but raises the question of how to obtain a non-trivial upper bound. Clearly, we cannot hope that the cost of every feasible solution will always be within some reasonable factor of the cost of a single item. However, in some cases it is enough to obtain an upper bound only for *minimal solutions*. A *minimal solution* is a feasible solution that is minimal with respect to set inclusion, i.e., a feasible solution all of whose proper subsets are not feasible. Minimal solutions arise naturally in the context of *covering* problems, which are the problems for which feasible solutions have the property of being monotone inclusion-wise, that is, the property that adding items to a feasible solution cannot render it infeasible. (For example, adding a set to a set cover yields a set cover, so *set cover* is a covering problem. In contrast, adding an edge to a spanning tree does not yield a tree, so *minimum spanning tree* is not a covering problem.) The idea of focusing on minimal solutions leads to the following definition.

Definition 2 *Given a set of constraints $\mathcal{F}$ on vectors in $\mathbb{R}^n$ and a number $r \geq 1$, a weight vector $w \in \mathbb{R}^n$ is said to be* r-effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all minimal feasible solutions x.*

Note that any fully r-effective weight function is also r-effective, while the opposite direction is not always true.

If we can prove that our algorithm uses r-effective weight functions and returns minimal solutions, we will have essentially proved that it is an r-approximation algorithm. Designing an algorithm to output minimal solutions is not hard. Most of the creative effort is therefore concentrated in finding an r-effective weight function (for a small r).

In this section we present local ratio algorithms for the *partial cover* and *feedback vertex set* problems. Both algorithms depend on obtaining minimal solutions. We describe and analyze the algorithm for *partial set cover* in full detail. We then outline a general local ratio framework for covering problems, and discuss the algorithm for *feedback vertex set* informally with reference to this framework.

4.1. Partial Set Cover

In the *partial set cover* problem the input consists of a collection of non-empty sets $\mathcal{C} = \{S_1, \ldots, S_n\}$, a weight function w on the sets, and a number k. The objective is to find a minimum-cost sub-collection of $\mathcal{C}$ that covers at least k elements in $U \triangleq \bigcup_{i=1}^{n} S_i$. We assume that a feasible solution exists, i.e., that $k \leq |U|$. The *partial cover* problem generalizes *set cover* since in the set cover problem k is simply set to $|U|$.

Next, we present a $\max\{d_{\max}, 2\}$-approximation algorithm. (Recall that $d_{\max} = \max_{x \in U} \deg(x)$, where $\deg(x) = |\{S \in \mathcal{C} \mid x \in S\}|$.)

Algorithm PSC($U, \mathcal{C}, w, k$)

1. If $k \leq 0$, return $\emptyset$.
2. Else, if there exists a set $S \in \mathcal{C}$ such that $w(S) = 0$ do:
3. Let $U', \mathcal{C}'$ be the instance obtained by removing S.
4. $P' \leftarrow$ **PSC($U', \mathcal{C}', w, k - |S|$)**.
5. If P' covers at least k elements in U:
6. Return the solution $P = P'$.
7. Else:
8. Return the solution $P = P' \cup \{S\}$.
9. Else:
10. Let ϵ be maximal such that $\epsilon \cdot \min\{|S|, k\} \leq w(S)$ for all $S \in \mathcal{C}$.
11. Define the weight functions $w_1(x) = \epsilon \cdot \min\{|S|, k\}$ and $w_2 = w - w_1$.
12. Return **PSC($U, \mathcal{C}, w_2, k$)**.

Note the slight abuse of notation in Line 4. The weight function in the recursive call is not w itself, but rather the restriction of w to $\mathcal{C}'$. We will continue to silently abuse notation in this manner.

Let us analyze the algorithm. We claim that the algorithm finds a minimal solution that is $\max\{d_{\max}, 2\}$-approximate. Intuitively, Lines 3–8 ensure that the solution

returned is minimal, while the weight decomposition in Lines 9–12 ensures that every minimal solution is $\max\{d_{\max}, 2\}$-approximate. This is done by associating with each set S a weight that is proportional to its "covering power," which is the number of elements in S, but not more than k, since covering more than k elements is no better than covering k elements.

Proposition 3 *Algorithm **PSC** returns a feasible minimal solution.*

Proof. The proof is by induction on the recursion. At the recursion basis the solution returned is the empty set, which is both feasible (since $k \leq 0$) and minimal. For the inductive step, $k > 0$ and there are two cases to consider. If Lines 9–12 are executed, then the solution returned is feasible and minimal by the inductive hypothesis. Otherwise, Lines 3–8 are executed. By the inductive hypothesis P' is minimal and feasible with respect to $(U', \mathcal{C}', k - |S|)$. If $P' = \emptyset$ then $|S| \geq k$ and $P = P' \cup \{S\}$ is clearly feasible and minimal. Otherwise, P' covers at least $k - |S|$ elements in $U \setminus S$, and by minimality, for all $T \in P'$, the collection $P' \setminus \{T\}$ covers less than $k - |S|$ elements that are not contained in S. (Note that $P' \neq \emptyset$ implies $k > |S|$.) Consider the solution P. Either $P = P'$, which is the case if P' covers at least k or more elements, or else $P = P' \cup \{S\}$, in which case P covers at least $k - |S|$ elements that are not contained in S and an additional $|S|$ elements that are contained in S. In either case P covers at least k elements and is therefore feasible. It is also minimal (in either case), since for all $T \in P$, if $T \neq S$, then $P \setminus \{T\}$ covers less than $k - |S|$ elements that are not contained in S and at most $|S|$ elements that are contained in S, for a total of less than k elements, and if $T = S$, then $S \in P$, which implies $P = P' \cup \{S\}$, which is only possible if $P \setminus \{S\} = P'$ covers less than k elements. ■

Proposition 4 *The weight function w_1 used in Algorithm **PSC** is $\max\{d_{max}, 2\}$-effective.*

Proof. In terms of w_1, every feasible solution costs at least $\epsilon \cdot k$, since it either contains a set whose cost is $\epsilon \cdot k$, or else consists solely of sets whose size is less than k, in which case the cost of the solution equals ϵ times the total sizes of the sets in the solution, which must be at least k in order to cover k elements. To prove that every minimal solution costs at most $\epsilon \cdot k \cdot \max\{d_{\max}, 2\}$, consider a non-empty minimal feasible solution P. If P is a singleton, its cost is at most ϵk, and the claim follows. Otherwise, we prove the claim by showing that $\sum_{S \in P} |S| \leq k \cdot \max\{d_{\max}, 2\}$. We say that an element x is *covered r times* by P if $|\{S \in P \mid x \in S\}| = r$. We bound the total number of times elements are covered by P, since this number is equal to $\sum_{S \in P} |S|$. Clearly every element x may be covered at most $\deg(x) \leq d_{\max}$ times. Thus, if t is the number of elements that are covered by P twice or more, these elements contribute at most $t \cdot d_{\max}$ to the count. As for the elements that are covered only once, they contribute exactly $\sum_{S \in P} |S|_1$, where $|S|_1$ is the number of elements covered by S but not by any other member of P. Let $S^* = \arg\min\{|S|_1 \mid S \in P\}$. Then (by the choice of S^* and the fact that P is not a singleton) $|S^*|_1 \leq \sum_{S \in P \setminus \{S^*\}} |S|_1$. In addition, $t + \sum_{S \in P \setminus \{S^*\}} |S|_1 < k$ by the minimality of P. Thus the elements that are covered only

once contribute $|S^*|_1 + \sum_{S \in P \setminus \{S^*\}} |S|_1 \leq 2 \sum_{S \in P \setminus \{S^*\}} |S|_1 < 2(k-t)$, and the total is less than $t \cdot d_{\max} + 2(k-t) \leq k \cdot \max\{d_{\max}, 2\}$, where the inequality follows from the fact that $t \leq k$ (which is implied by the minimality of P). ■

Theorem 5 *Algorithm* ***PSC*** *returns* $\max\{d_{max}, 2\}$*-approximate solutions.*

Proof. The proof is by induction on the recursion. In the base case the solution returned is the empty set, which is optimal. For the inductive step, if Lines 3–8 are executed, then P' is $\max\{d_{\max}, 2\}$-approximate with respect to $(U', \mathcal{C}', w, k - |S|)$ by the inductive hypothesis. Since $w(S) = 0$, the cost of P equals that of P' and the optimum for $(U, \mathcal{C}, w, k)$ cannot be smaller than the optimum value for $(U', \mathcal{C}', w, k - |S|)$ because if P^* is an optimal solution for $(U, \mathcal{C}, w, k)$, then $P^* \setminus \{S\}$ is a feasible solution of the same cost for $(U', \mathcal{C}', w, k - |S|)$. Hence P is $\max\{d_{\max}, 2\}$-approximate with respect to $(U, \mathcal{C}, w, k)$. If, on the other hand, Lines 10–12 are executed, then by the inductive hypothesis the solution returned is $\max\{d_{\max}, 2\}$-approximate with respect to w_2, and by Proposition 4 it is also $\max\{d_{\max}, 2\}$-approximate with respect to w_1. Thus by the Local Ratio Theorem it is $\max\{d_{\max}, 2\}$-approximate with respect to w as well. ■

4.2. A Framework

A local ratio algorithm for a covering problem typically consists of a three-way *if* condition that directs execution to one of the following three primitives: *computation of optimal solution*, *problem size reduction*, or *weight decomposition*. The top-level description of the algorithm is:

1. If a zero-cost minimal solution can be found, do: *computation of optimal solution*.
2. Else, if the problem contains a zero-cost element, do: *problem size reduction*.
3. Else, do: *weight decomposition*.

The three types of primitives are:

Computation of optimal solution. Find a zero-cost minimal solution and return it. (Typically, the solution is simply the empty set.) This is the recursion basis.

Problem size reduction. This primitive consists of three parts.

1. Pick a zero-cost item, and assume it is taken into the solution. Note that this changes the problem instance, and may entail further changes to achieve consistency with the idea that the item has been taken temporarily to the solution. For example, in the *partial set cover* problem we selected a zero-cost set and assumed it is part of our solution. Hence, we have deleted all elements contained in it and reduced the covering requirement parameter k by the size of this set. Note that the modification of the problem instance may be somewhat more involved. For example, a graph edge might be eliminated by *contracting* it. As a result, two vertices are "fused" together and the edges

incident on them are merged or deleted. In other words, the modification may consist of removing some existing items and introducing new ones. This, in turn, requires that the weight function be modified to cover the new items. However, it is important to realize that the modification of the weight function amounts to a re-interpretation of the old weights in terms of the new instance and not to an actual change of weights.

2. Solve the problem recursively on the modified instance.
3. If the solution returned (when re-interpreted in terms of the original instance) is feasible (for the original instance), return it. Otherwise, add the deleted zero-cost item to the solution, and return it.

Weight decomposition. Construct an r-effective weight function w_1 such that $w_2 = w - w_1$ is non-negative, and solve the problem recursively using w_2 as the weight function. Return the solution obtained.

We note that the above description of the framework should not be taken too literally. Each branch of the three-way *if* statement may actually consist of several sub-cases, only one of which is to be executed. We shall see an example of this in the algorithm for *feedback vertex set* (Section 4.3).

The analysis of an algorithm that follows the framework is similar to our analysis for *partial set cover*. It consists of proving the following three claims.

Claim 1. The algorithm outputs a minimal feasible solution.
This claim is proven by induction on the recursion. In the base case the solution is feasible and minimal by design. For the inductive step, if the algorithm performs *weight decomposition*, then by the inductive hypothesis the solution is feasible and minimal. If the algorithm performs *problem size reduction*, the claim follows from the fact that the solution returned by the recursive call is feasible and minimal with respect to the modified instance (by the inductive hypothesis) and it is extended only if it is infeasible with respect to the current instance. Although the last argument seems straightforward, the details of a rigorous proof tend to be slightly messy, since they depend on the way in which the instance is modified.

Claim 2. The weight function w_1 is r-effective.
The proof depends on the combinatorial structure of the problem at hand. Indeed, the key to the design of a local ratio algorithm is understanding the combinatorial properties of the problem and finding the "right" r and r-effective weight function (or functions).

Claim 3. The algorithm computes an r-approximate solution.
The proof of this claim is also by induction on the recursion, based on the previous two claims. In the base case the *computation of optimal solution* ensures that the solution returned is r-approximate. For the inductive step, we have two options. In the *problem size reduction* case, the solution found recursively for the modified instance is r-approximate by the inductive hypothesis, and it has the same cost as the solution generated for the original instance, since the two solutions may only differ by a zero-weight item. This, combined with the fact that the optimum can

only decrease because of instance modification, yields the claim. In the *weight decomposition* case, the claim follows by the inductive hypothesis and the Local Ratio Theorem, since the solution is feasible and minimal, and w_1 is r-effective.

4.3. Feedback Vertex Set

A set of vertices in an undirected graph is called a *feedback vertex set* (FVS for short) if its removal leaves an acyclic graph (i.e., a forest). Another way of saying this is that the set intersects all cycles in the graph. The *feedback vertex set* problem is: given a vertex-weighted graph, find a minimum-weight FVS. In this section we describe and analyze a 2-approximation algorithm for the problem following our framework for covering problems.

The algorithm is as follows.

Algorithm FVS(G, w)

1. If G is empty, return $\emptyset$.
2. If there exists a vertex $v \in V$ such that $\deg(v) \leq 1$ do:
3. return **FVS**$(G \setminus \{v\}, w)$.
4. Else, if there exists a vertex $v \in V$ such that $w(v) = 0$ do:
5. $F' \leftarrow$ **FVS**$(G \setminus \{v\}, w)$.
6. If F' is an FVS with respect to G:
7. Return F'.
8. Else:
9. Return $F = F' \cup \{v\}$.
10. Else:
11. Let $\epsilon = \min_{v \in V} \frac{w(v)}{\deg(v)}$.
12. Define the weight functions $w_1(v) = \epsilon \cdot \deg(v)$ and $w_2 = w - w_1$.
13. Return **FVS**(G, w_2)

The analysis follows the pattern outlined above—the only interesting part is showing that w_1 is 2-effective. For a given set of vertices X let us denote $\deg(X) = \sum_{v \in X} \deg(v)$. Since $w_1(F) = \epsilon \cdot \deg(F)$ for any FVS F, it is sufficient to demonstrate the existence of a number b such that for all minimal solutions F, $b \leq \deg(F) \leq 2b$. Note that the weight decomposition is only applied to graphs in which all vertices have degree at least 2, so we shall henceforth assume that our graph $G = (V, E)$ is such a graph.

Consider a minimal feasible solution F. The removal of F from G leaves a forest on $|V| - |F|$ nodes. This forest contains less than $|V| - |F|$ edges, and thus the number of edges deleted to obtain it is greater than $|E| - (|V| - |F|)$. Since each of these edges is incident on some vertex in F, we get $|E| - (|V| - |F|) < \deg(F)$ (which

is true even if F is not minimal). Let F^* be a minimum cardinality FVS, and put $b = |E| - (|V| - |F^*|)$. Then $b < \deg(F)$, and it remains to prove that $\deg(F) \leq 2b = 2|E| - 2(|V| - |F^*|)$. We show equivalently that $\deg(V \setminus F) \geq 2(|V| - |F^*|)$. To do so we select for each vertex $v \in F$ a cycle C_v containing v but no other member of F (the minimality of F ensures the existence of such a cycle). Let P_v denote the path obtained from C_v by deleting v, and let V' denote the union of the vertex sets of the P_vs. Consider the connected components of the subgraph induced by V'. Each connected component fully contains some path P_v. Since the cycle C_v must contain a member of the FVS F^*, either P_v contains a vertex of $F^* \setminus F$ or else $v \in F^* \cap F$. Thus there are at most $|F^* \setminus F|$ connected components that contain vertices of F^* and at most $|F^* \cap F|$ that do not, for a total of at most F^* connected components. Hence, the subgraph contains at least $|V'| - |F^*|$ edges, all of which have both endpoints in V'. In addition, G contains at least $2|F|$ edges with exactly one endpoint in V'—the two edges connecting each $v \in F$ with the path P_v (to form the cycle C_v). It follows that $\deg(V') \geq 2(|V'| - |F^*|) + 2|F|$. Thus, bearing in mind that $F \subseteq V \setminus V'$ and that the degree of every vertex is at least two, we see that

$$\begin{aligned}\deg(V \setminus F) &= \deg(V') + \deg((V \setminus V') \setminus F) \\ &\geq 2(|V'| - |F^*|) + 2|F| + 2(|V| - |V'| - |F|) \\ &= 2(|V| - |F^*|).\end{aligned}$$

4.4. Background

Partial set cover. The *partial set cover* problem was first studied by Kearns [48] in relation to learning. He proves that the performance ratio of the greedy algorithm is at most $2H_n + 3$, where n is the number of sets. (Recall that H_n is the nth harmonic number.) Slavik [60] improves this bound to H_k. The special case in which the cardinality of every set is exactly 2 is called the *partial vertex cover* problem. This problem was studied by Bshouty and Burroughs [25], who obtained the first polynomial time 2-approximation algorithm for it. The max $\{d_{\max}, 2\}$-approximation algorithm for *partial set cover* given in this section (Algorithm **PSC**) is due to Bar-Yehuda [14]. In fact, his approach can be used to approximate an extension of the partial cover problem in which there is a *length* l_i associated with each element x, and the goal is to cover elements of total length at least k. (The plain *set cover* problem is the special case where $l_i = 1$ for all i.) Gandhi et al. [34] present a multi-phase primal-dual algorithm for *partial cover* achieving a performance ratio of max $\{d_{\max}, 2\}$.

Minimal solutions and feedback vertex set. Minimal solutions first appeared in a local ratio algorithm for FVS. FVS is NP-hard [47] and MAX SNP-hard [53], and at least as hard to approximate as vertex cover [51]. An $O(\log n)$-approximation algorithm for unweighted FVS is implies by a lemma due to Erdös and Pósa [31]. Monien and Shultz [55] improve the ratio to $\sqrt{\log n}$. Bar-Yehuda et al. [18] present a 4-approximation algorithm for unweighted FVS, and an $O(\log n)$-approximation algorithm for weighted FVS. Bafna, Berman, and Fujito [9] present a local ratio

2-approximation algorithm for weighted FVS, whose weight decomposition is somewhat similar to the one used in Algorithm **FVS**. Their algorithm is the first local ratio algorithm to make use of minimal solutions (although this concept was used earlier in primal-dual algorithms for *network design* problems [58, 1, 37]). At about the same time, Becker and Geiger [20] also obtained a 2-approximation algorithm for FVS. Algorithm **FVS** is a recursive local ratio formulation of their algorithm. Chudak et al. [26] suggest yet another 2-approximation algorithm, and give primal-dual analyses of the three algorithms. Fujito [33] proposes a generic local ratio algorithm for a certain type of node-deletion problems. Algorithm **RecursiveVC**, Algorithm **FVS**, and the algorithm from [9] can be seen as applications of Fujito's generic algorithm. Bar-Yehuda [15] presents a unified local ratio approach for covering problems. He present a short generic approximation algorithm which can explain many known optimization and approximation algorithms for covering problems (including Fujito's generic algorithm). Later, Bar-Yehuda and Rawitz [19] devised a framework that extends the one from [15]. The notion of effectiveness of a weight function first appeared in [15]. The corresponding primal-dual notion appeared earlier in [22].

5. Scheduling Problems

In this section we turn to applications of the local ratio technique in the context of resource allocation and scheduling problems. Resource allocation and scheduling problems are immensely popular objects of study in the field of approximation algorithms and combinatorial optimization, owing to their direct applicability to many real-life situations and their richness in terms of mathematical structure. Historically, they were among the first to be analyzed in terms of worst-case approximation ratio, and research into these problems continues actively to this day.

In very broad terms, a scheduling problem is one in which *jobs* presenting different demands vie for the use of some limited *resource*, and the goal is to resolve all conflicts. Conflicts are resolved by scheduling different jobs at different times and either enlarging the amount of available resource to accommodate all jobs or accepting only a subset of the jobs. Accordingly, we distinguish between two types of problems. The first type is when the resource is fixed but we are allowed to reject jobs. The problem is then to maximize the number (or total weight) of accepted jobs, and there are two natural ways to measure the quality of a solution: in *throughput maximization* problems the measure is the total weight of accepted jobs (which we wish to maximize), and in *loss minimization* problems it is the total weight of rejected jobs (which we wish to minimize). While these two measures are equivalent in terms of optimal solutions, they are completely distinct when one considers approximate solutions. The second type of problem is *resource minimization*. Here we must satisfy all jobs, and can achieve this by increasing the amount of resource. The objective is to minimize the cost of doing so.

Our goal in this section is twofold. First, we demonstrate the applicability of the local ratio technique in an important field of research, and second, we take the

opportunity of tackling throughput maximization problems to develop a local ratio theory for maximization problems in general. We begin with the latter. We present a local ratio theorem for maximization problems and sketch a general framework based on it in Section 5.1. We apply this framework to a collection of throughput maximization problems in Section 5.2. Following that, we consider loss minimization in Section 5.3, and resource minimization in Section 5.4.

5.1. Local Ratio for Maximization Problems

The Local Ratio Theorem for maximization problems is nearly identical to its minimization counterpart. It applies to optimization problems that can be formulated as follows.

> Given a *weight vector* $w \in \mathbb{R}^n$ and a set of *feasibility constraints* $\mathcal{F}$, find a *solution vector* $x \in \mathbb{R}^n$ satisfying the constraints in $\mathcal{F}$ that maximizes the inner product $w \cdot x$.

Before stating the Local Ratio Theorem for maximization problems, we remind the reader of our convention that a feasible solution to a maximization problem is said to be r-approximate if its weight is at least $1/r$ times the optimum weight (so approximation factors are always greater than or equal to 1).

Theorem 5 (Local Ratio—Maximization Problems) *Let $\mathcal{F}$ be a set of feasibility constraints on vectors in $\mathbb{R}^n$. Let $w, w_1, w_2 \in \mathbb{R}^n$ be such that $w = w_1 + w_2$. Let $x \in \mathbb{R}^n$ be a feasible solution (with respect to $\mathcal{F}$) that is r-approximate with respect to w_1 and with respect to w_2. Then, x is r-approximate with respect to w as well.*

Proof. Let x^*, x_1^*, and x_2^* be optimal solutions with respect to w, w_1, and w_2, respectively. Clearly, $w_1 \cdot x_1^* \geq w_1 \cdot x^*$ and $w_2 \cdot x_2^* \geq w_2 \cdot x^*$. Thus, $w \cdot x = w_1 \cdot x + w_2 \cdot x \geq \frac{1}{r}(w_1 \cdot x_1^*) + \frac{1}{r}(w_2 \cdot x_2^*) \geq \frac{1}{r}(w_1 \cdot x^*) + \frac{1}{r}(w_2 \cdot x^*) = \frac{1}{r}(w \cdot x^*)$. ■

The general structure of a local ratio approximation algorithm for a maximization problem is similar to the one described for the minimization case in Section 4.2. It too consists of a three-way *if* condition that directs execution to one of three main options: *optimal solution*, *problem size reduction*, or *weight decomposition*. There are several differences though. In contrast to what is done in the minimization case, we make no effort to keep the weight function non-negative, i.e., in *weight decomposition* steps we allow w_2 to take on negative values. Also, in *problem size reduction* steps we usually remove an element whose weight is either zero or negative. Finally and most importantly, we strive to construct *maximal* solutions rather than minimal ones. This affects our choice of w_1 in *weight decomposition* steps. The weight function w_1 is chosen such that every *maximal* solution (a feasible solution that cannot be extended) is r-approximate with respect to it[2]. In accordance, when the recursive call returns in

[2] We actually impose a somewhat weaker condition, as described in Section 5.2.

problem size reduction steps, we extend the solution if possible (rather than if necessary), but we attempt to do so only for zero-weight elements (not negative weight ones).

As in the minimization case, we use the notion of effectiveness.

Definition 3 *In the context of maximization problems, a weight function w is said to be r-*effective *if there exists a number b such that $b \leq w \cdot x \leq r \cdot b$ for all maximal feasible solutions x.*

5.2. Throughput Maximization Problems

Consider the following general problem. The input consists of a set of *activities*, each requiring the utilization of a given limited *resource*. The amount of resource available is fixed over time; we normalize it to unit size for convenience. The activities are specified as a collection of sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Each set represents a single activity: it consists of all possible *instances* of that activity. An instance $I \in \mathcal{A}_i$ is defined by the following parameters.

1. A half-open time interval $[s(I), e(I))$ during which the activity will be executed. We call $s(I)$ and $e(I)$ the *start-time* and the *end-time* of the instance.
2. The amount of resource required for the activity. We refer to this amount as the *width* of the instance and denote it $d(I)$. Naturally, $0 < d(I) \leq 1$.
3. The *weight* $w(I) \geq 0$ of the instance. It represents the profit to be gained by scheduling this instance of the activity.

Different instances of the same activity may have different parameters of duration, width, or weight. A *schedule* is a collection of instances. It is *feasible* if (1) it contains at most one instance of every activity, and (2) for all time instants t, the total width of the instances in the schedule whose time interval contains t does not exceed 1 (the amount of resource available). The goal is to find a feasible schedule that maximizes the total weight of instances in the schedule. For example, consider the problem instance depicted in Figure 1. The inputconsists of three activities, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, each comprising several instances, depicted as rectangles in the respective rows. The projection of each rectangle on the t axis represents the corresponding instance's time interval. The height of each rectangle represents the resource requirement (the instance's width) on a 1:5 scale (e.g., the leftmost instance of activity $\mathcal{C}$ has width 0.6). The weights of the instances are not shown. Numbering the instances of each activity from left to right, the schedule $\{\mathcal{A}(1), \mathcal{C}(3), \mathcal{C}(4)\}$ is infeasible because activity $\mathcal{C}$ is scheduled twice; $\{\mathcal{A}(1), \mathcal{C}(1)\}$ is infeasible because both instances overlap and their total width is 1.4; $\{\mathcal{A}(1), \mathcal{B}(1), \mathcal{C}(4)\}$ is feasible.

In the following sections we describe local ratio algorithms for several special cases of the general problem. We use the following notation. For a given activity instance I, $\mathcal{A}(I)$ denotes the activity to which I belongs and $\mathcal{I}(I)$ denotes the set of all activity instances that intersect I but belong to activities other than $\mathcal{A}(I)$.

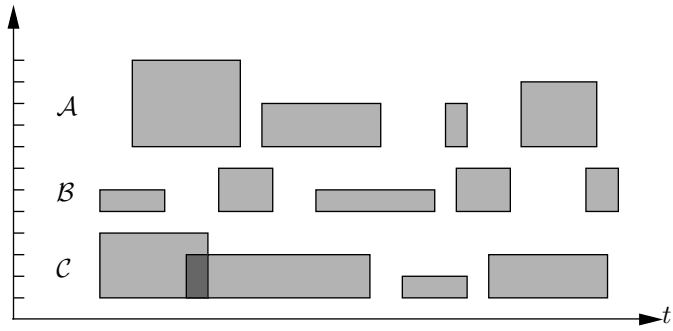


Figure 1. An example of activities and instances.

5.2.1. Interval Scheduling

In the *interval scheduling* problem we must schedule jobs on a single processor with no preemption. Each job consists of a finite collection of time intervals during which it may be scheduled. The problem is to select a maximum weight subset of non-conflicting intervals, at most one interval for each job. In terms of our general problem, this is simply the special case where every activity consists of a finite number of instances and the width of every instance is 1.

To design the weight decomposition for this problem, we examine the properties of maximal schedules. Let J be the activity instance with minimum end-time among all activity instances of all activities (breaking ties arbitrarily). The choice of J ensures that all of the intervals intersecting it intersect each other (see Figure 2). Consider a maximal schedule $\mathcal{S}$. Clearly $\mathcal{S}$ cannot contain more than one instance from $\mathcal{A}(J)$, nor can it contain more than one instance from $\mathcal{I}(J)$, since all of these instances intersect each other. Thus $\mathcal{S}$ contains at most two intervals from $\mathcal{A}(J) \cup \mathcal{I}(J)$. On the other hand, $\mathcal{S}$ must contain at least one instance from $\mathcal{A}(J) \cup \mathcal{I}(J)$, for otherwise it would not be maximal (since J could be added to it). This implies that the weight function

$$w_1(I) = \epsilon \cdot \begin{cases} 1 & I \in \mathcal{A}(J) \cup \mathcal{I}(J), \\ 0 & \text{otherwise}, \end{cases}$$

is 2-effective for any choice of $\epsilon > 0$, and we can expect to obtain a 2-approximation algorithm based on it.

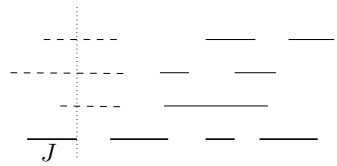


Figure 2. J, $\mathcal{A}(J)$, and $\mathcal{I}(J)$: heavy lines represent $\mathcal{A}(J)$; dashed lines represent $\mathcal{I}(J)$.

A logical course of action is to fix $\epsilon = \min\{w(I) : I \in \mathcal{A}(J) \cup \mathcal{I}(J)\}$ and to solve the problem recursively on $w - w_1$, relying on two things: (1) w_1 is 2-effective; and (2) the solution returned is maximal. However, we prefer a slightly different approach. We show that w_1 actually satisfies a stronger property than 2-effectiveness. For a given activity instance I, we say that a feasible schedule is *I-maximal* if either it contains I, or it does not contain I but adding I to it will render it infeasible. Clearly, every maximal schedule is also I-maximal for any given I, but the converse is not necessarily true. The stronger property satisfied by the above w_1 is that every J-maximal schedule is 2-approximate with respect to w_1 (for all $\epsilon > 0$). To see this, observe that no optimal schedule may contain more than two activity instances from $\mathcal{A}(J) \cup \mathcal{I}(J)$, whereas every J-maximal schedule must contain at least one (if it contains none, it cannot be J-maximal since J can be added). The most natural choice of ϵ is $\epsilon = w(J)$.

Our algorithm for *interval scheduling* is based on the above observations. The initial call is **IS**$(\mathcal{A}, w)$, where $\mathcal{A}$ is the set of jobs, which we also view as the set of all $\cup_{i=1}^{m} \mathcal{A}_i$.

Algorithm IS$(\mathcal{A}, w)$

1. If $\mathcal{A} = \emptyset$, return $\emptyset$.
2. If there exists an interval I such that $w(I) \leq 0$ do:
3. Return **IS**$(\mathcal{A} \setminus \{I\}, w)$.
4. Else:
5. Let J be the instance with minimum end-time in $\mathcal{A}$.
6. Define $w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J) \cup \mathcal{I}(J), \\ 0 & \text{otherwise}, \end{cases}$
   and let $w_2 = w - w_1$.
7. $\mathcal{S}' \leftarrow$ **IS**$(\mathcal{A}, w_2)$.
8. If $\mathcal{S}' \cup \{J\}$ is feasible:
9. Return $\mathcal{S} = \mathcal{S}' \cup \{J\}$.
10. Else:
11. Return $\mathcal{S} = \mathcal{S}'$.

As with similar previous claims, the proof that Algorithm **IS** is 2-approximation is by induction on the recursion. At the basis of the recursion (Line 1) the schedule returned is optimal and hence 2-approximate. For the inductive step there are two possibilities. If the recursive call is made in Line 3, then by the inductive hypothesis the schedule returned is 2-approximate with respect to $(\mathcal{A} \setminus \{I\}, w)$, and since the weight of I is non-positive, the optimum for $(\mathcal{A}, w)$ cannot be greater than the optimum for $(\mathcal{A} \setminus \{I\}, w)$. Thus the schedule returned is 2-approximate with respect to $(\mathcal{A}, w)$ as well. If the recursive call is made in Line 7, then by the inductive hypothesis $\mathcal{S}'$ is 2-approximate with respect to w_2, and since $w_2(J) = 0$ and $S \subseteq S' \cup \{J\}$, it follows that $\mathcal{S}$ too is 2-approximate with respect to w_2. Since $\mathcal{S}$ is J-maximal by construction (Lines 8–11), it is also 2-approximate with respect to w_1. Thus, by the Local Ratio Theorem, it is 2-approximate with respect to w as well.

5.2.2. *Independent Set in Interval Graphs*

Consider the special case of *interval scheduling* in which each activity consists of a single instance. This is exactly the problem of finding a maximum weight independent set in an interval graph (each instance corresponds to an interval), and it is well known that this problem can be solved optimally in polynomial time (see, e.g., [39]). We claim that Algorithm **IS** solves this problem optimally too, and to prove this it suffices to show that every J-maximal solution is optimal with respect to w_1. This is so because at most one instance from $\mathcal{A}(J) \cup \mathcal{I}(J)$ may be scheduled in any feasible solution (since $\mathcal{A}(J) = \{J\}$), and every J-maximal solution schedules one.

5.2.3. *Scheduling on Parallel Identical Machines*

In this problem the resource consists of k parallel identical machines. Each activity instance may be assigned to any of the k machines. Thus $d(I) = 1/k$ for all I.

In order to approximate this problem we use Algorithm **IS**, but with a different choice of w_1, namely,

$$w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J), \\ 1/k & I \in \mathcal{I}(J), \\ 0 & \text{otherwise.} \end{cases}$$

The analysis of the algorithm is similar to the one used for the case $k = 1$ (i.e., interval scheduling). It suffices to show that every J-maximal schedule is 2-approximate with respect to w_1. This is so because every J-maximal schedule either contains an instance from $\mathcal{A}(J)$ or a set of instances intersecting J that prevent J from being added to the schedule. In the former case, the weight of the schedule with respect to w_1 is at least $w(J)$. In the latter case, since k machines are available but J cannot be added, the schedule must already contain k activity instances from $\mathcal{I}(J)$, and its weight (with respect to w_1) is therefore at least $k \cdot \frac{1}{k} \cdot w(J) = w(J)$. Thus the weight of every J-maximal schedule is at least $w(J)$. On the other hand, an optimal schedule may contains at most one instance from $\mathcal{A}(J)$ and at most k instances from $\mathcal{I}(J)$ (as they all intersect each other), and thus its weight cannot exceed $w(J) + k \cdot \frac{1}{k} \cdot w(J) = 2w(J)$.

Remark. Our algorithm only finds a set of activity instances that can be scheduled, but does not construct an actual assignment of instances to machines. This can be done easily by scanning the instances (in the solution found by the algorithm) in increasing order of end-time, and assigning each to an arbitrary available machine. It is easy to see that such a machine must always exist. Another approach is to solve the problem as a special case of scheduling on parallel unrelated machines (described next).

5.2.4. *Scheduling on Parallel Unrelated Machines*

Unrelated machines differ from identical machines in that a given activity instance may be assignable only to a subset of the machines, and furthermore, the profit derived

from scheduling it on a machine may depend on the machine (i.e., it need not be the same for all allowable machines). We can assume that each activity instance may be assigned to precisely one machine. (Otherwise, simply replicate each instance once for each allowable machine.) We extend our scheduling model to handle this problem as follows. We now have k types of unit quantity resource (corresponding to the k machines), each activity instance specifies the (single) resource type it requires, and the feasibility constraint applies to each resource type separately. Since no two instance may be processed concurrently on the same machine, we set $d(I) = 1$ for all instances I.

To approximate this problem, we use Algorithm **IS** but define $\mathcal{I}(J)$ slightly differently. Specifically, $\mathcal{I}(J)$ is now defined as the set of instances intersecting J that belong to other activities *and can be scheduled on the same machine as* J. It is easy to see that again, every J-maximal schedule is 2-approximate with respect to w_1, and thus the algorithm is 2-approximation.

5.2.5. Bandwidth Allocation of Sessions in Communication Networks

Consider a scenario in which the bandwidth of a communication channel must be allocated to sessions. Here the resource is the channel's bandwidth, and the activities are sessions to be routed through the channel. A session is specified as a list of intervals in which it can be scheduled, together with a width requirement and a weight for each interval. The goal is to find the most profitable set of sessions that can utilize the available bandwidth.

To approximate this problem we first consider the following two special cases.

Special Case 1 All instances are *wide*, i.e., $d(I) > 1/2$ for all I.
Special Case 2 All activity instances are *narrow*, i.e., $d(I) \leq 1/2$ for all I.

In the case of wide instances the problem reduces to interval scheduling since no pair of intersecting instances may be scheduled together. Thus, we use Algorithm **IS** to find a 2-approximate schedule with respect to the wide instances only.

In the case of *narrow* instances we find a 3-approximate schedule by a variant of Algorithm **IS** in which w_1 is defined as follows:

$$w_1(I) = w(J) \cdot \begin{cases} 1 & I \in \mathcal{A}(J), \\ 2 \cdot d(I) & I \in \mathcal{I}(J), \\ 0 & \text{otherwise.} \end{cases}$$

To prove that the algorithm is a 3-approximation algorithm it suffices to show that every J-maximal schedule is 3-approximate with respect to w_1. (All other details are essentially the same as for interval scheduling.) A J-maximal schedule either contains an instance of $\mathcal{A}(J)$ or contains a set of instances intersecting J that prevent J from being added to the schedule. In the former case the weight of the schedule is at least $w(J)$.

In the latter case, since J cannot be added, the combined width of activity instances from $\mathcal{I}(J)$ in the schedule must be greater than $1 - d(J) \geq 1/2$, and thus their total weight (with respect to w_1) must be greater than $\frac{1}{2} \cdot 2w(J) = w(J)$. Thus, the weight of every J-maximal schedule is at least $w(J)$. On the other hand, an optimal schedule may contain at most one instance from $\mathcal{A}(J)$ and at most a set of instances from $\mathcal{I}(J)$ with total width 1 and hence total weight $2w(J)$. Thus, the optimum weight is at most $3w(J)$, and therefore every J-maximal schedule is 3-approximate with respect to w_1.

In order to approximate the problem in the general case where both narrow and wide activity instances are present, we solve it separately for the narrow instances and for the wide instances, and return the solution of greater weight. Let OPT be the optimum weight for all activity instances, and let OPT_n and OPT_w be the optimum weight for the narrow instance and for the wide instances, respectively. Then, the weight of the schedule found is at least $\max\left\{\frac{1}{3}OPT_n, \frac{1}{2}OPT_w\right\}$. Now, either $OPT_n \geq \frac{3}{5}OPT$, or else $OPT_w \geq \frac{2}{5}OPT$. In either case the schedule returned is 5-approximate.

5.2.6. Continuous Input

In our treatment of the above problems we have tacitly assumed that each activity is specified as a finite set of instances. We call this type of input *discrete input*. In a generalization of the problem we can allow each activity to consist of infinitely many instances by specifying the activity as a finite collection of *time windows*. A *time window* $\mathcal{T}$ is defined by four parameters: *start-time* $s(\mathcal{T})$, *end-time* $e(\mathcal{T})$, *instance length* $l(\mathcal{T}) \leq e(\mathcal{T}) - s(\mathcal{T})$, and *weight* $w(\mathcal{T})$. It represents the set of all instances defined by intervals of length $l(\mathcal{T})$ contained in the interval $[s(\mathcal{T}), e(\mathcal{T}))$ with associated profit $w(\mathcal{T})$. We call this type of input *continuous input*. The ideas underlying our algorithms for discrete input apply equally well to continuous input, and we can achieve the same approximation guarantees. However, because infinitely many intervals are involved, the running times of the algorithms might become super-polynomial (although they are guaranteed to be finite). To obtain efficiency we can sacrifice an additive term of ϵ in the approximation guarantee in return for an implementation whose worst case time complexity is $O(n^2/\epsilon)$. Reference [11] contains the full details.

5.2.7. Throughput Maximization with Batching

The main constraint in many scheduling problems is that no two jobs may be scheduled on the same machine at the same time. However, there are situations in which this constraint is relaxed, and *batching* of jobs is allowed. Consider, for example, a multimedia-on-demand system with a fixed number of channels through which video films are broadcast to clients (e.g., through a cable TV network). Each client requests a particular film and specifies several alternative times at which he or she would like to view it. If several clients wish to view the same movie at the same time, their requests can be *batched* together and satisfied simultaneously by a single transmission. In the throughput maximization version of this problem, we aim to maximize the revenue by deciding which films to broadcast, and when, subject to the constraint that the number of channels is fixed and only a single movie may be broadcast on a given channel at any time.

Formally, the batching problem is defined as follows. We are given a set of jobs, to be processed on a system of parallel identical machines. Each job is defined by its *type*, its *weight*, and a set of *start-times*. In addition, each job type has a *processing time* associated with it. (In terms of our video broadcasting problem, machines correspond to channels, jobs correspond to clients, job types correspond to movies, job weights correspond to revenues, start-times correspond to alternative times at which clients wish to begin watching the films they have ordered, and processing times correspond to movie lengths.) A *job instance* is a pair (J, t) where J is a job and t is one of its start-times. Job instance (J, t) is said to *represent* job J. We associate with it the time interval $[t, t + p)$, where p is the processing time associated with J's type. A *batch* is a set of job instances such that all jobs represented in the set are of identical type, no job is represented more than once, and all time intervals associated with the job instances are identical. We associate with the batch the time interval associated with its job instances. Two batches *conflict in jobs* if there is a job represented in both; they *conflict in time* if their time intervals are not disjoint. A *feasible schedule* is an assignment of batches to machines such that no two batches in the schedule conflict in jobs and no two batches conflicting in time are assigned to the same machine. The goal is to find a maximum-weight feasible schedule. (The weight of a schedule is the total weight of jobs represented in it.)

The batching problem can be viewed as a variant of the scheduling problem we have been dealing with up till now. For simplicity, let us consider the single machine case. For every job, consider the set of batches in which it is represented. All of these batches conflict with each other, and we may consider them instances of a single activity. In addition, two batches with conflicting time intervals also conflict with each other, so it seems that the problem reduces to *interval scheduling*. There is a major problem with this interpretation, though, even disregarding the fact that the number of batches may be exponential. The problem is that if activities are defined as we have suggested, i.e., each activity is the set of all batches containing a particular job, then activities are not necessarily disjoint sets of instances, and thus they lack a property crucial for our approach to *interval scheduling*. Nevertheless, the same basic approach can be still be applied, though the precise details are far too complex to be included in a survey such as this. We refer the reader to Bar-Noy et al. [12], who describe a 4-approximation algorithm for *bounded* batching (where there is an additional restriction that no more than a fixed number of job instances may be batched together), and a 2-approximation algorithm for unbounded batching.

5.3. Loss Minimization

In the previous section we dealt with scheduling problems in which our aim was to maximize the profit from scheduled jobs. In this section we turn to the dual problem of minimizing the loss due to rejected jobs.

Recall that in our general scheduling problem (defined in Section 5.2) we are given a limited resource, whose amount is fixed over time, and a set of activities requiring the utilization of this resource. Each activity is a set of instances, at most one of which

is to be selected. In the loss minimization version of the problem considered here, we restrict each activity to consist of a single instance, but we allow the amount of resource to vary in time. Thus the input consists of the activity specification as well as a positive function $D(t)$ specifying the amount of resource available at every time instant t. Accordingly, we allow arbitrary positive instance widths (rather than assuming that all widths are bounded by 1). A schedule is feasible if for all time instants t, the total width of instances in the schedule containing t is at most $D(t)$. Given a feasible schedule, the feasible solution it defines is the set of all activity instances *not* in the schedule. The goal is to find a minimum weight feasible solution.

We now present a variant of Algorithm **IS** achieving an approximation guarantee of 4. We describe the algorithm as one that finds a feasible schedule, with the understanding that the feasible solution actually being returned is the schedule's complement. The algorithm is as follows. Let $(\mathcal{A}, w)$ denote the input, where $\mathcal{A}$ is the description of the activities excluding their weights, and w is the weight function. If the set of all activity instances in $\mathcal{A}$ constitutes a feasible schedule, return this schedule. Otherwise, if there is a zero-weight instance I, delete it, solve the problem recursively to obtain a schedule $\mathcal{S}$, and return either $\mathcal{S} \cup \{I\}$ or $\mathcal{S}$, depending (respectively) on whether $\mathcal{S} \cup \{I\}$ is a feasible schedule or not. Otherwise, decompose w by $w = w_1 + w_2$ (as described in the next paragraph), where $w_2(I) \geq 0$ for all activity instances I, with equality for at least one instance, and solve recursively for $(\mathcal{A}, w_2)$.

Let us define the decomposition of w by showing how to compute w_1. For a given time instant t, let $\mathcal{I}(t)$ be the set of activity instances containing t. Define $\Delta(t) = \sum_{I \in \mathcal{I}(t)} d(I) - D(t)$. To compute w_1, find t^* maximizing $\Delta(\cdot)$ and let $\Delta^* = \Delta(t^*)$. Assuming $\Delta^* > 0$ (otherwise the schedule containing all instances is feasible), let

$$w_1(I) = \epsilon \cdot \begin{cases} \min\{\Delta^*, d(I)\} & I \in \mathcal{I}(t^*), \\ 0 & \text{otherwise}, \end{cases}$$

where ϵ (which depends on t^*) is the unique scaler resulting in $w_2(I) \geq 0$ for all I and $w_2(I) = 0$ for at least one I. A straightforward implementation of this algorithm runs in time polynomial in the number of activities and the number of time instants at which $D(t)$ changes value.

To prove that the algorithm is 4-approximation, it suffices (by the usual arguments) to show that w_1 is 4-effective. In other words, it suffices to show that every solution defined by a maximal feasible schedule is 4-approximate with respect to w_1. In the sequel, when we say *weight*, we mean weight with respect to w_1.

Observation 6 *Every collection of instances from $\mathcal{I}(t^*)$ whose total width is at least Δ^* has total weight at least $\epsilon\Delta^*$.*

Observation 7 *Both of the following evaluate to at most $\epsilon\Delta^*$: (1) the weight of any single instance, and (2) the total weight of any collection of instances whose total width is at most Δ^*.*

Consider an optimal solution (with respect to w_1). Its weight is the total weight of instances in $\mathcal{I}(t^*)$ that are not in the corresponding feasible schedule. Since all of these instances intersect at t^*, their combined width must be at least Δ^*. Thus, by Observation 6, the optimal weight is at least $\epsilon\Delta^*$. Now consider the complement of a maximal feasible schedule $\mathcal{M}$. We claim that it is 4-approximate because its weight does not exceed $4\epsilon\Delta^*$. To prove this, we need the following definitions. For a given time instant t, let $\overline{\mathcal{M}}(t)$ be the set of instances containing t that are not in the schedule $\mathcal{M}$, (i.e., $\overline{\mathcal{M}}(t) = \mathcal{I}(t) \setminus \mathcal{M}$). We say that t is *critical* if there is an instance $I \in \overline{\mathcal{M}}(t)$ such that adding I to $\mathcal{M}$ would violate the width constraint at t. We say that t is critical *because of* I. Note that a single time instant may be critical because of several different activity instances.

Lemma 8 *If t is a critical time instant, then $\sum_{I\in\overline{\mathcal{M}}(t)} w_1(I) < 2\epsilon\Delta^*$.*

Proof. Let J be an instance of maximum width in $\overline{\mathcal{M}}(t)$. Then, since t is critical, it is surely critical because of J. This implies that $\sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I) > D(t) - d(J)$. Thus,

$$\begin{aligned}\sum_{I\in\overline{\mathcal{M}}(t)} d(I) &= \sum_{I\in\mathcal{I}(t)} d(I) - \sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I)\\ &= D(t) + \Delta(t) - \sum_{I\in\mathcal{M}\cap\mathcal{I}(t)} d(I)\\ &< d(J) + \Delta(t)\\ &\leq d(J) + \Delta^*.\end{aligned}$$

Hence, $\sum_{I\in\overline{\mathcal{M}}(t)\setminus\{J\}} d(I) < \Delta^*$, and therefore, by Observation 7,

$$\sum_{I\in\overline{\mathcal{M}}(t)} w_1(I) = \sum_{I\in\overline{\mathcal{M}}(t)\setminus\{J\}} w_1(I) + w_1(J) \leq \epsilon\Delta^* + \epsilon\Delta^* = 2\epsilon\Delta^*.$$

■

Thus there are two cases to consider. If t^* is a critical point, then the weight of the solution is $\sum_{I\in\overline{\mathcal{M}}(t^*)} w_1(I) < 2\epsilon\Delta^*$ and we are done. Otherwise, let $t_L < t^*$ and $t_R > t^*$ be the two critical time instants closest to t^* on both sides (it may be that only one of them exists). The maximality of the schedule implies that every instance in $\overline{\mathcal{M}}(t^*)$ is the cause of criticality of at least one time instant. Thus each such instance must contain t_L or t_R (or both). It follows that $\overline{\mathcal{M}}(t^*) \subseteq \overline{\mathcal{M}}(t_L) \cup \overline{\mathcal{M}}(t_R)$. Hence, by Lemma 8, the total weight of these instances is less than $4\epsilon\Delta^*$.

5.3.1. Application: General Caching

In the *general caching* problem a replacement schedule is sought for a cache that must accommodate pages of varying sizes. The input consists of a fixed cache size $D > 0$, a collection of pages $\{1, 2, \ldots, m\}$, and a sequence of n requests for pages.

Each page j has a *size* $0 < d(j) \leq D$ and a weight $w(j) \geq 0$, representing the cost of loading it into the cache. We assume for convenience that time is discrete and that the ith request is made at time i. (These assumptions cause no loss of generality as will become evident from our solution.) We denote by $r(i)$ the page being requested at time i. A *replacement schedule* is a specification of the contents of the cache at all times. It must satisfy the following condition. For all $1 \leq i \leq n$, page $r(i)$ is present in the cache at time i and the sum of sizes of the pages in the cache at that time is not greater than the cache size D. The initial contents of the cache (at time 0) may be chosen arbitrarily. Alternatively, we may insist that the cache be empty initially. The weight of a given replacement schedule is $\sum w(r(i))$ where the sum is taken over all i such that page $r(i)$ is absent from the cache at time $i - 1$. The objective is to find a minimum weight replacement schedule.

Observe that if we have a replacement schedule that evicts a certain page at some time between two consecutive requests for it, we may as well evict it immediately after the first of these requests and bring it back only for the second request. Thus, we may restrict our attention to schedules in which for every two consecutive requests for a page, either the page remains present in the cache at all times between the first request and the second, or it is absent from the cache at all times in between. This leads naturally to a description of the problem in terms of time intervals, and hence to a reduction of the problem to our loss minimization problem, as follows. Given an instance of the general caching problem, define the resource amount function by $D(i) = D - d(r(i))$ for $1 \leq i \leq n$, and $D(0) = D$ (or $D(0) = 0$ if we want the cache to be empty initially). Define the activity instances as follows. Consider the request made at time i. Let j be the time at which the previous request for $r(i)$ is made, or $j = -1$ if no such request is made. If $j + 1 \leq i - 1$, we define an activity instance with time interval $[j + 1, i - 1]$, weight $w(r(i))$, and width $d(r(i))$. This reduction implies a 4-approximation algorithm for the general caching problem via our 4-approximation algorithm for loss minimization.

5.4. Resource Minimization

Until now we have dealt with scheduling problems in which the resource was limited, and thus we were allowed to schedule only a subset of the jobs. In this section we consider the case where all jobs must be scheduled, and the resource is not limited (but must be paid for). The objective is to minimize the cost of the amount of resource in the solution. We present a 3-approximation algorithm for such a problem. We refer to the problem as the *bandwidth trading* problem, since it is motivated by bandwidth trading in next generation networks. (We shall not discuss this motivation here, as it is rather lengthy.)

The algorithm we present here is somewhat unusual in that it does not use an r-effective weight function in the weight decomposition steps. Whereas previous algorithms prune (or extend), if possible, the solution returned by the recursive call in order

to turn it into a "good" solution, i.e., one that is minimal (or maximal), the algorithm we present here uses a weight function for which good solutions are solutions that satisfy a certain property different from minimality or maximality. Accordingly, it modifies the solution returned in a rather elaborate manner.

5.4.1. Bandwidth Trading

In the *bandwidth trading* problem we are given a set of machine *types* $\mathcal{T} = \{T_1, \ldots, T_m\}$ and a set of *jobs* $J = \{1, \ldots, n\}$. Each machine type T_i is defined by two parameters: a time interval $I(T_i)$ during which it is *available*, and a weight $w(T_i)$, which represents the cost of allocating a machine of this type. Each job j is defined by a single time interval $I(j)$ during which it must be processed. We say that job j *contains* time t if $t \in I(j)$. A given job j may be *scheduled feasibly* on a machine of type T if type T is available throughout the job's interval, i.e., if $I(j) \subseteq I(T)$. A *schedule* is a set of machines together with an assignment of each job to one of them. It is *feasible* if every job is assigned feasibly and no two jobs with intersecting intervals are assigned to the same machine. The cost of a feasible schedule is the total cost of the machines it uses, where the cost of a machine is defined as the weight associated with its type. The goal is to find a minimum-cost feasible schedule. We assume that a feasible schedule exists. (This can be checked easily.)

Our algorithm for the problem follows.

Algorithm BT$(\mathcal{T}, J, w)$

1. If $J = \emptyset$, return $\emptyset$.
2. Else, if there exists a machine type $T \in \mathcal{T}$ such that $w(T) = 0$ do:
3. Let J' be the set of jobs that can be feasibly scheduled on machines of type T, i.e., $J' = \{j \in J \mid I(j) \subseteq I(T)\}$.
4. $S' \leftarrow$ **BT**$(\mathcal{T} \setminus \{T\}, J \setminus J', w)$.
5. Extend S' to all J by allocating $|J'|$ machines of type T and scheduling one job from J' on each.
6. Return the resulting schedule S.
7. Else:
8. Let t be a point in time contained in a maximum number of jobs, and let $\mathcal{T}_t$ be the set of machine types available at time t (see Figure 3).
9. Let $\epsilon = \min\{w(T) \mid T \in \mathcal{T}_t\}$.
10. Define the weight functions $w_1(T) = \begin{cases} \epsilon & T \in \mathcal{T}_t, \\ 0 & \text{otherwise,} \end{cases}$ and $w_2 = w - w_1$.
11. $S' \leftarrow$ **BT**$(\mathcal{T}, J, w_2)$.
12. Transform S' into a new schedule S (in a manner described below.
13. Return S.

To complete the description of the algorithm we must describe the transformation of S' to S referred to in Line 12. We shall do so shortly, but for now let us just point out two facts relating to the transformation.

1. For all machine types T, S does not use more machines of type T than S'.
2. Let k be the number of jobs containing time t (Line 8). The number of machines used by S whose types are in $\mathcal{T}_t$ is at most $3k$.

Based on these facts, we now prove that the algorithm is a 3-approximation algorithm. The proof is by induction on the recursion. In the base case ($J = \emptyset$), the schedule returned is optimal and therefore 3-approximate. For the inductive step there are two cases. If the recursive invocation is made in Line 4, then by the inductive hypothesis, S' is 3-approximate with respect to $(\mathcal{T} \setminus \{T\}, J \setminus J', w)$. In addition, no job in $J \setminus J'$ can be scheduled feasibly on a machine of type T; all jobs in J' can be scheduled feasibly on machines of type T; and machines of type T are free ($w(T) = 0$). Thus $w(S) = w(S')$, and the optimum cost for $(\mathcal{T}, J, w)$ is the same as for $(\mathcal{T} \setminus \{T\}, J \setminus J', w)$. Therefore S is 3-approximate. The second case in the inductive step is that the recursive call is made in Line 11. In this case, S' is 3-approximate with respect to w_2 by the inductive hypothesis. By the first fact above, $w_2(S) \leq w_2(S')$, and therefore S too is 3-approximate with respect to w_2. By the second fact above, $w_1(S) \leq 3k\epsilon$, and because there are k jobs containing time t—each of which can be scheduled only on machines whose types are in $\mathcal{T}_t$, and no two of which may be scheduled on the same machine—the optimum cost is at least $k\epsilon$. Thus S is 3-approximate with respect to w_1. By the Local Ratio Theorem, S is therefore 3-approximate with respect to w.

It remains to describe the transformation of S' to S in Line 12. Let $J_t \subseteq J$ be the set of jobs containing time t, and recall that $k = |J_t|$. An example (with $k = 3$) is given in Figure 3. The strips above the line represent jobs, and those below the line represent machine types. The darker strips represent the jobs in J_t and the machine types in $\mathcal{T}_t$. Let $\mathcal{M}_t \subseteq \mathcal{M}_S$ be the set of machines that are available at time t and are used by S, and let $J_{\mathcal{M}_t}$ be the set of jobs scheduled by S on machines in $\mathcal{M}_t$. ($J_{\mathcal{M}_t}$ consists of the jobs J_t and possibly additional jobs.) If $|\mathcal{M}_t| \leq 3k$ then $S' = S$. Otherwise, we choose a subset of $\mathcal{M}'_t \subseteq \mathcal{M}_t$ of size at most $3k$ and reschedule all of the jobs in $J_{\mathcal{M}_t}$ on these machines. The choice of $\mathcal{M}'_t$ and the construction of S' are as follows.

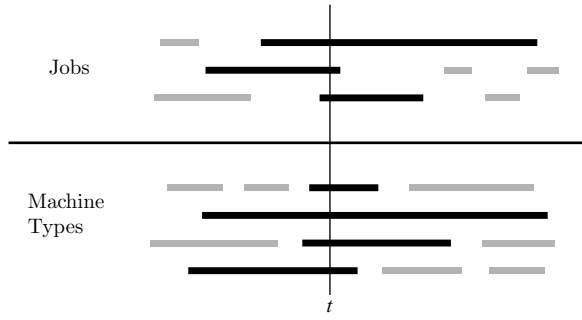


Figure 3. Jobs containing time t (top, dark), and machine types available at time t (bottom, dark).

1. Let $\mathcal{M}_c \subseteq \mathcal{M}_t$ be the set of k machines to which the k jobs in J_t are assigned. (Each job must be assigned to a different machine since they all exist at time t). Let J_c be the set of all jobs scheduled on machines in $\mathcal{M}_c$. We schedule these jobs the same as in S.
2. Let $\mathcal{M}_l \subseteq \mathcal{M}_t \setminus \mathcal{M}_c$ be the set of k machines in $\mathcal{M}_t \setminus \mathcal{M}_c$ with leftmost left endpoints. (See example in Figure 4.) Let $J_l \subseteq J_{\mathcal{M}_t}$ be the set of jobs in $J_{\mathcal{M}_t}$ that lie completely to the left of time point t and are scheduled by S on machines in $\mathcal{M}_t \setminus \mathcal{M}_c$. We schedule these jobs on machines from $\mathcal{M}_l$ as follows. Let t' be the rightmost left endpoint of a machine in $\mathcal{M}_l$. The jobs in J_l that contain t' must be assigned in S to machines from $\mathcal{M}_l$. We retain their assignment. We proceed to schedule the remaining jobs in J_l greedily by order of increasing left endpoint. Specifically, for each job j we select any machine in $\mathcal{M}_l$ on which we have not already scheduled a job that conflicts with j and schedule j on it. This is always possible since all k machines are available between t' and t, and thus if a job cannot be scheduled, its left endpoint must be contained in k other jobs that have already been assigned. These $k+1$ jobs coexist at the time instant defining the left endpoint of the job that cannot be assigned, in contradiction with the fact that k is the maximal number jobs coexisting at a any time.

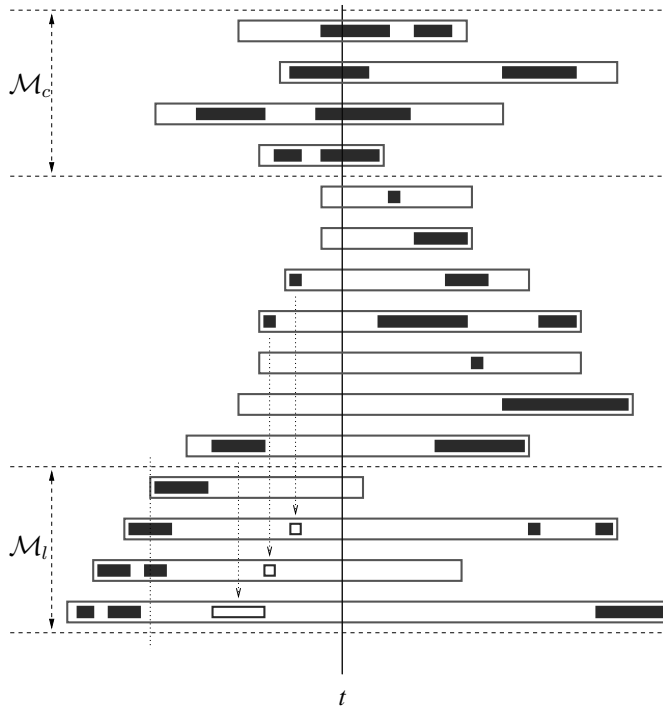


Figure 4. Rescheduling jobs that were assigned to $\mathcal{M}_t$ and exist before time t.

3. Let $\mathcal{M}_r \subseteq \mathcal{M}_t \setminus \mathcal{M}_c$ be the set of k machines in $\mathcal{M}_t \setminus \mathcal{M}_c$ with rightmost right endpoints. ($\mathcal{M}_r$ and $\mathcal{M}_l$ and not necessarily disjoint.) Let $J_r \subseteq J_{\mathcal{M}_t}$ be the set of jobs in $J_{\mathcal{M}_t}$ that lie completely to the right of time point t and are scheduled by S on machines in $\mathcal{M}_t \setminus \mathcal{M}_c$. We schedule these jobs on machines from $\mathcal{M}_r$ in a similar manner to the above.

We have thus managed to schedule $J_c \cup J_l \cup J_r = J_{\mathcal{M}_t}$ on no more than $3k$ machines from $\mathcal{M}_t$, as desired.

5.5. Background

Throughput maximization. Single machine scheduling with one instance per activity is equivalent to *maximum weight independent set in interval graphs* and hence polynomial-time solvable [39]. Arkin and Silverberg [3] solve the problem efficiently even for unrelated multiple machines. The problem becomes NP-hard (even in the single machine case) if multiple instances per activity are allowed [61] (i.e., the problem is *interval scheduling*) or if instances may require arbitrary amounts of the resource. (In the latter case the problem is NP-hard as it contains *knapsack* [35] as a special case in which all time intervals intersect.) Spieksma [61] studies the unweighted interval scheduling problem. He proves that it is Max-SNP-hard, and presents a simple greedy 2-approximation algorithm. Bar-Noy et al. [13] consider *real-time scheduling*, in which each job is associated with a release time, a deadline, a weight, and a processing time on each of the machines. They give several constant factor approximation algorithms for various variants of the throughput maximization problem. They also show that the problem of scheduling unweighted jobs on unrelated machine is Max-SNP-hard.

Bar-Noy et al. [11], present a general framework for solving resource allocation and scheduling problems that is based on the local ratio technique. Given a resource of fixed size, they present algorithms that approximate the maximum throughput or the minimum loss by a constant factor. The algorithms apply to many problems, among which are: real-time scheduling of jobs on parallel machines; bandwidth allocation for sessions between two endpoints; general caching; dynamic storage allocation; and bandwidth allocation on optical line and ring topologies. In particular, they improve most of the results from [13] either in the approximation factor or in the running time complexity. Their algorithms can also be interpreted within the primal-dual schema (see also [19]) and are the first local ratio (or primal-dual) algorithms for a maximization problems. Sections 5.2 and 5.3, with the exception of Section 5.2.7, are based on [11].

Independently, Berman and DasGupta [21] also improve upon the algorithms given in [13]. They develop an algorithm for interval scheduling that is nearly identical to the one from [11]. Furthermore, they employ the same rounding idea used in [11] in order to contend with time windows. In addition to single machine scheduling, they also consider scheduling on parallel machines, both identical and unrelated.

Chuzhoy et al. [27] consider the unweighted *real-time scheduling* problem and present an $(e/(e-1)+\epsilon)$-approximation algorithm. They generalize this algorithm to achieve a ratio of $(1+e/(e-1)+\epsilon)$ for unweighted *bandwidth allocation*.

The batching problem discussed in Section 5.2.7 is from Bar-Noy et al. [12]. In the case of bounded batching, they describe a 4-approximation algorithm for discrete input and a $(4+\epsilon)$-approximation algorithm for continuous input. In the case on unbounded batching, their approximation factors are 2 and $2+\epsilon$, respectively. However, in the discrete input case the factor 2 is achieved under an additional assumption. (See [12] for more details.) In the *parallel batch processing* model all jobs belong to the same family, and any group of jobs can be batched together. A batch is completed when the largest job in the batch is completed. This model was studied by Brucker et al. [24] and by Baptiste [10]. The model discussed in [12] is called *batching with incompatible families*. This model was studied previously with different objective functions such as *weighted sum of completion times* [62, 30, 8] and *total tardiness* [54].

Loss minimization. Section 5.3 is based on [11]. Albers et al. [2] consider the *general caching* problem. They achieve a "pseudo" O(1)-approximation factor (using LP rounding) by increasing the size of the cache by O(1) times the size of the largest page, i.e., their algorithm finds a solution using the enlarged cache whose cost is within a constant factor of the optimum for for the original cache size. When the cache size may not be increased, they achieve an $O(\log(M+C))$ approximation factor, where M and C denote the cache size and the largest page reload cost, respectively. The reduction of *general caching* to the loss minimization problem discussed in Section 5.3 is also from [2].

Resource minimization. Section 5.4 is based on a write-up by Bhatia et al. [23]. The general model of the resource minimization problem, where the sets of machine types on which a job can be processed are arbitrary, is essentially equivalent (approximation-wise) to *set cover* [23, 45]. Kolen and Kroon [49, 50] show that versions of the general problem considered in [23] are NP-hard.

Acknowledgments

We thank the anonymous reviewer for many corrections and suggestions. One of the suggestions was a proof of the 2-effectiveness of the weight function used in the context of *feedback vertex set* (Section 4.3) which was considerably simpler than our original proof. This proof inspired the proof that now appears in the text.

References

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24(3): 440–456 (1995).

[2] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, (1999) pp. 31–40.

[3] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18: 1–8 (1987).

[4] S. Arora and C. Lund. Hardness of approximations. In Hochbaum [44], chapter 10, pp. 399–446.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3): 501–555 (1998).

[6] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1): 70–122 (1998).

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation; Combinatorial optimization problems and their approximability properties*. Springer Verlag, (1999).

[8] M. Azizoglu and S. Webster. Scheduling a batch processing machine with incompatible job families. *Computer and Industrial Engineering*, 39(3–4): 325–335 (2001).

[9] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics*, 12(3): 289–297 (1999).

[10] P. Baptiste. Batching identical jobs. *Mathematical Methods of Operations Research*, 52(3): 355–367 (2000).

[11] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Shieber. A unified approach to approximating resource allocation and schedualing. *Journal of the ACM*, 48(5): 1069–1090 (2001).

[12] A. Bar-Noy, S. Guha, Y. Katz, J. Naor, B. Schieber, and H. Shachnai. Throughput maximization of real-time scheduling with batching. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms*, (2002) pp. 742–751.

[13] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing*, 31(2): 331–352 (2001).

[14] Bar-Yehuda. Using homogeneous weights for approximating the partial cover problem. *Journal of Algorithms*, 39(2): 137–144 (2001).

[15] R. Bar-Yehuda. One for the price of two: A unified approach for approximating covering problems. *Algorithmica*, 27(2): 131–144 (2000).

[16] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2): 198–203 (1981).

[17] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46 (1985).

[18] R. Bar-Yehuda, D. Geiger, J. Naor, and R. M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM Journal on Computing*, 27(4): 942–959 (1998).

[19] R. Bar-Yehuda and D. Rawitz. On the equivalence between the primal-dual schema and the local-ratio technique. In *4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, volume 2129 of *LNCS*, (2001) pp. 24–35.

[20] A. Becker and D. Geiger. Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *Artificial Intelligence*, 83(1): 167–188 (1996).

[21] P. Berman and B. DasGupta. Multi-phase algorithms for throughput maximization for real-time scheduling. *Journal of Combinatorial Optimization*, 4(3): 307–323 (2000).

[22] D. Bertsimas and C. Teo. From valid inequalities to heuristics: A unified view of primal-dual approximation algorithms in covering problems. *Operations Research*, 46(4): 503–514 (1998).

[23] R. Bhatia, J. Chuzhoy, A. Freund, and J. Naor. Algorithmic aspects of bandwidth trading. In *30th International Colloquium on Automata, Languages, and Programming*, volume 2719 of *LNCS*, (2003) pp. 751–766.

[24] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. N. Potts, T. Tautenhahn, and S. L. van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1(1): 31–54 (1998).

[25] N. H. Bshouty and L. Burroughs. Massaging a linear programming solution to give a 2-approximation for a generalization of the vertex cover problem. In *15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *LNCS*, pp. 298–308. Springer, (1998).

[26] F. A. Chudak, M. X. Goemans, D. S. Hochbaum, and D. P. Williamson. A primal-dual interpretation of recent 2-approximation algorithms for the feedback vertex set problem in undirected graphs. *Operations Research Letters*, 22: 111–118 (1998).

[27] J. Chuzhoy, R. Ostrovsky, and Y. Rabani. Approximation algorithms for the job interval selection problem and related scheduling problems. In *42nd IEEE Symposium on Foundations of Computer Science*, (2001) pp. 348–356.

[28] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3): 233–235 (1979).

[29] I. Dinur and S. Safra. The importance of being biased. In *34th ACM Symposium on the Theory of Computing*, (2002) pp. 33–42.

[30] G. Dobson and R. S. Nambimadom. The batch loading and scheduling problem. *Operations Research*, 49(1): 52–65 (2001).

[31] P. Erdös and L. Pósa. On the maximal number of disjoint circuits of a graph. *Publ. Math. Debrecen*, 9: 3–12 (1962).

[32] U. Feige. A threshold of $\ln n$ for approximating set cover. In *28th Annual Symposium on the Theory of Computing*, (1996) pp. 314–318.

[33] T. Fujito. A unified approximation algorithm for node-deletion problems. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 86: 213–231 (1998).

[34] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. In *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *LNCS*, (2001) pp. 225–236.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, (1979).

[36] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1: 237–267 (1976).

[37] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2): 296–317 (1995).

[38] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Hochbaum [44], chapter 4, pp. 144–191.

[39] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, (1980). Second edition: *Annuals of Discrete Mathematics,* 57, Elsevier, Amsterdam (2004).

[40] E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms*, (2000) pp. 329–337.

[41] J. Håstad. Some optimal inapproximability results. In *29th Annual ACM Symposium on the Theory of Computing*, (1997) pp. 1–10.

[42] D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11(3): 555–556 (1982).

[43] D. S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6: 243–254 (1983).

[44] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problem*. PWS Publishing Company, (1997).

[45] K. Jansen. An approximation algorithm for the license and shift class design problem. *European Journal of Operational Research*, 73: 127–131 (1994).

[46] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9: 256–278 (1974).

[47] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pp. 85–103, New York, (1972). Plenum Press.

[48] M. Kearns. *The Computational Complexity of Machine Learning*. M.I.T. Press, (1990).

[49] A. W. J. Kolen and L. G. Kroon. On the computational complexity of (maximum) class scheduling. *European Journal of Operational Research*, 54: 23–38 (1991).

[50] A. W. J. Kolen and L. G. Kroon. An analysis of shift class design problems. *European Journal of Operational Research*, 79: 417–430 (1994).

[51] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary problems is NP-complete. *Journal of Computer and System Sciences*, 20: 219–230 (1980).

[52] L. Lovász. On the ratio of optimal integral and fractional covers. *Discreate Mathematics*, 13: 383–390 (1975).

[53] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In *20th International Colloquium on Automata, Languages and Programming*, volume 700 of *LNCS*, July (1993) pp. 40–51.

[54] S. V. Mehta and R. Uzsoy. Minimizing total tardiness on a batch processing machine with incompatable job famalies. *IIE Transactions*, 30(2): 165–178 (1998).

[55] B. Monien and R. Shultz. Four approximation algorithms for the feedback vertex set problem. In *7th conference on graph theoretic concepts of computer science*, (1981) pp. 315–390.

[56] B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22: 115–123 (1985).

[57] G. L. Nemhauser and L. E. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8: 232–248 (1975).

[58] R. Ravi and P. Klein. When cycles collapse: A general approximation technique for constrained two-connectivity problems. In *3rd Conference on Integer Programming and Combinatorial Optimization*, (1993) pp. 39–56.

[59] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *29th ACM Symposium on the Theory of Computing*, (1997) pp. 475–484.

[60] P. Slavík. Improved performance of the greedy algorithm for partial cover. *Information Processing Letters*, 64(5): 251–254 (1997).

[61] F. C. R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2(5): 215–227 (1999).

[62] R. Uzsoy. Scheduling batch processing machines with incompatible job families. *International Journal of Production Research*, 33: 2685–2708 (1995).

6

Domination Analysis of Combinatorial Optimization Algorithms and Problems

Gregory Gutin
Department of Computer Science
Royal Holloway
University of London, Egham
Surrey TW20 0EX, UK

Anders Yeo
Department of Computer Science
Royal Holloway
University of London, Egham
Surrey TW20 0EX, UK

Abstract

We provide an overview of an emerging area of domination analysis (DA) of combinatorial optimization algorithms and problems. We consider DA theory and its relevance to computational practice.

1. Introduction

In the recently published book [19], Chapter 6 is partially devoted to domination analysis (DA) of the Traveling Salesman Problem (TSP) and its heuristics. The aim of this chapter is to provide an overview of the whole area of DA. In particular, we describe results that significantly generalize the corresponding results for the TSP.

To make reading of this chapter more active, we provide questions that range from simple to relatively difficult ones. Also, we add research questions that supply the interested reader with open and challenging problems.

This chapter is organized as follows. In Subsection 1.1 of this section we motivate the use of DA in combinatorial optimization. We provide a short introduction to DA in Subsection 1.2. We conclude this section by giving additional terminology and notation.

One of the goals of DA is to analyze the domination number or domination ratio of various algorithms. Domination number (ratio) of a heuristic H for a combinatorial optimization problem P is the maximum number (fraction) of all solutions that are not better than the solution found by H for any instance of P of size n. In Section 2 we consider TSP heuristics of large domination number. In Subsection 2.1 we provide a theorem that allows one to prove that a certain Asymmetric TSP heuristic is of very large domination number. We also provide an application of the theorem. In Subsection 2.2 we show how DA can be used in analysis of local search heuristics. Upper bounds for the domination numbers of Asymmetric TSP heuristics are derived in Subsection 2.3.

Section 3 is devoted to DA for other optimization problems. We demonstrate that problems such as the Minimum Partition, Max Cut, Max k-SAT and Fixed Span Frequency Assignment admit polynomial time algorithms of large domination number. On the other hand, we prove that some other problems including the Maximum Clique and the Minimum Vertex Cover do not admit algorithms of relatively large domination ratio unless P = NP.

Section 4 shows that, in the worst case, the greedy algorithm obtains the unique worst possible solution for a wide family of combinatorial optimization problems and, thus, in the worst case, the greedy algorithm is no better than the random choice for such problems. We conclude the chapter by a short discussion of DA practicality.

1.1. Why Domination Analysis?

Exact algorithms allow one to find optimal solutions to NP-hard combinatorial optimization (CO) problems. Many research papers report on solving large instances of some NP-hard problems (see, e.g., Chapters 2 and 4 in [19]). The running time of exact algorithms is often very high for large instances, and very large instances remain beyond the capabilities of exact algorithms.

Even for instances of moderate size, if we wish to remain within seconds or minutes rather than hours or days of running time, only heuristics can be used. Certainly, with heuristics, we are not guaranteed to find optimum, but good heuristics normally produce near-optimal solutions. This is enough in most applications since very often the data and/or mathematical model are not exact anyway.

Research on CO heuristics has produced a large variety of heuristics especially for well-known CO problems. Thus, we need to choose the best ones among them. In most of the literature, heuristics are compared in computational experiments. While experimental analysis is of definite importance, it cannot cover all possible families of

instances of the CO problem at hand and, in particular, it normally does not cover the hardest instances.

Approximation Analysis [4] is a frequently used tool for theoretical evaluation of CO heuristics. Let $\mathcal{H}$ be a heuristic for a combinatorial minimization problem P and let $\mathcal{I}_n$ be the set of instances of P of size n. In approximation analysis, we use the performance ratio $r_{\mathcal{H}}(n) = \max\{f(I)/f^*(I) : I \in \mathcal{I}_n\}$, where $f(I)$ ($f^*(I)$) is the value of the heuristic (optimal) solution of I. Unfortunately, for many CO problems, estimates for $r_{\mathcal{H}}(n)$ are not constants and provide only a vague picture of the quality of heuristics.

Domination Analysis (DA) provides an alternative and a complement to approximation analysis. In DA, we are interested in the domination number or domination ratio of heuristics (these parameters have been defined earlier). In many cases, DA is very useful. For example, we will see in Section 4 that the greedy algorithm has domination number 1 for many CO problems. In other words, the greedy algorithm, in the worst case, produces the unique worst possible solution. This is in line with latest computational experiments with the greedy algorithm, see, e.g., [28], where the authors came to the conclusion that the greedy algorithm 'might be said to self-destruct' and that it should not be used even as 'a general-purpose starting tour generator'.

The *Asymmetric Traveling Salesman Problem (ATSP)* is the problem of computing a minimum weight tour (Hamilton cycle) passing through every vertex in a weighted complete digraph on n vertices. See Figure 1. The *Symmetric TSP (STSP)* is the same problem, but on a complete undirected graph. When a certain fact holds for both ATSP and STSP, we will simply speak of *TSP*. Sometimes, the maximizing version of TSP is of interest, we denote it by *max TSP*.

APX is the class of CO problems that admit polynomial time approximation algorithms with a constant performance ratio [4]. It is well known that while max TSP belongs to APX, TSP does not. This is at odds with the simple fact that a 'good' approximation algorithm for max TSP can be easily transformed into an algorithm for

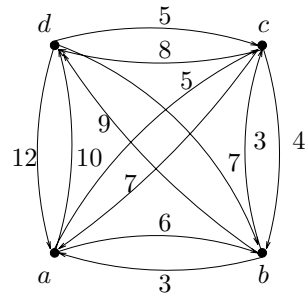


Figure 1. A complete weighted digraph.

TSP. Thus, it seems that both max TSP and TSP should be in the same class of CO problems. The above asymmetry was already viewed as a drawback of performance ratio already in the 1970's, see, e.g., [11, 30, 40]. Notice that from the DA point view max TSP and TSP are equivalent problems.

Zemel [40] was the first to characterize measures of quality of approximate solutions (of binary integer programming problems) that satisfy a few basic and natural properties: the measure becomes smaller for better solutions, it equals 0 for optimal solutions and it is the same for corresponding solutions of equivalent instances. While the performance ratio and even the relative error (see [4]) do not satisfy the last property, the parameter $1 - r$, where r is the domination ratio, does satisfy all of the properties.

Local Search (LS) is one of the most successful approaches in constructing heuristics for CO problems. Recently, several researchers started investigation of LS with Very Large Scale Neighbourhoods (see, e.g., [1, 12, 26]). The hypothesis behind this approach is that the larger the neighbourhood the better quality solution are expected to be found [1]. However, some computational experiments do not support this hypothesis, see, e.g., [15], where an LS with small neighbourhoods proves to be superior to that with large neighbourhoods. This means that some other parameters are responsible for the relative power of a neighbourhood. Theoretical and experimental results on TSP indicate that one such parameter may well be the domination ratio of the corresponding LS.

Sometimes, Approximation Analysis cannot be naturally used. Indeed, a large class of CO problems are multicriteria problems [14], which have several objective functions. (For example, consider STSP in which edges are assigned both time and cost, and one is required to minimize both time and cost.) We say that one solution s' of a multicriteria problems dominates another one s'' if the values of all objective functions at s' are not worse than those at s'' or the value of at least one objective function at s' is better than the value of the same objective function at s''. This definition allows us to naturally introduce the domination ratio (number) for multicriteria optimization heuristics. In particular, an algorithm that always finds a Pareto solution is of domination ratio 1.

In our view, it is advantageous to have bounds for both performance ratio and domination number (or, domination ratio) of a heuristic whenever it is possible. Roughly speaking this will enable us to see a 2D rather than 1D picture. For example, consider the double minimum spanning tree heuristic (DMST) for the Metric STSP (i.e., STSP with triangle inequality). DMST starts from constructing a minimum weight spanning tree T in the complete graph of the STSP, doubles every edge in T, finds a closed Euler trail E in the 'double' T, and cancels any repetition of vertices in E to obtain a TSP tour H. It is well-known and easy to prove that the weight of H is at most twice the weight of the optimal tour. Thus, the performance ratio for DMST is bounded by 2. However, Punnen, Margot and Kabadi [34] proved that the domination number of DMST is 1.

1.2. Introduction to Domination Analysis

Domination Analysis was formally introduced by Glover and Punnen [16] in 1997. Interestingly, important results on domination analysis for the TSP can be traced back to the 1970s, see Rublineckii [36] and Sarvanov [37].

Let $\mathcal{P}$ be a CO problem and let $\mathcal{H}$ be a heuristic for $\mathcal{P}$. The *domination number* domn($\mathcal{H}$,$\mathcal{I}$) of $\mathcal{H}$ *for a particular instance* $\mathcal{I}$ of $\mathcal{P}$ is the number of feasible solutions of $\mathcal{I}$ that are not better than the solution s produced by $\mathcal{H}$ including s itself. For example, consider an instance $\mathcal{T}$ of the STSP on 5 vertices. Suppose that the weights of tours in $\mathcal{T}$ are 3,3,5,6,6,9,9,11,11,12,14,15 (every instance of STSP on 5 vertices has 12 tours) and suppose that the greedy algorithm computes the tour T of weight 6. Then domn($\texttt{greedy}, \mathcal{T}$) = 9. In general, if domn($\mathcal{H}, \mathcal{I}$) equals the number of feasible solutions in $\mathcal{I}$, then $\mathcal{H}$ finds an optimal solution for $\mathcal{I}$. If domn($\mathcal{H}, \mathcal{I}$) = 1, then the solution found by $\mathcal{H}$ for $\mathcal{I}$ is the unique worst possible one.

The *domination number* domn($\mathcal{H}, n$) of $\mathcal{H}$ is the minimum of domn($\mathcal{H}, \mathcal{I}$) over all instances $\mathcal{I}$ of size n. Since the ATSP on n vertices has $(n-1)!$ tours, an algorithm for the ATSP with domination number $(n-1)!$ is exact. The domination number of an exact algorithm for the STSP is $(n-1)!/2$. If an ATSP heuristic $\mathcal{A}$ has domination number equal 1, then there is an assignment of weights to the arcs of each complete digraph K_n^*, $n \geq 2$, such that $\mathcal{A}$ finds the unique worst possible tour in K_n^*.

When the number of feasible solutions depends not only on the size of the instance of the CO problem at hand (for example, the number of independent sets of vertices in a graph G on n vertices depends on the structure of G), the domination ratio of an algorithm $\mathcal{A}$ is of interest: the *domination ratio* of $\mathcal{A}$, domr($\mathcal{A}, n$), is the minimum of domn($\mathcal{A}, \mathcal{I}$)/sol($\mathcal{I}$), where sol($\mathcal{I}$) is the number of feasible solutions of $\mathcal{I}$, taken over all instances $\mathcal{I}$ of size n. Clearly, domination ratio belongs to the interval [0, 1] and exact algorithms are of domination ratio 1.

The *Minimum Partition Problem* (MPP) is the following CO problem: given n nonnegative numbers $a_1, a_2, \ldots, a_n$, find a bipartition of the set $\{1, 2, \ldots, n\}$ into sets X and Y such that $d(X, Y) = |\sum_{i \in X} a_i - \sum_{i \in Y} a_i|$ is minimum. For simplicity, we assume that solutions X, Y and X', Y' are different as long as $X \neq X'$, i.e. even if $X = Y'$ (no symmetry is taken into consideration). Thus, the MPP has 2^n solutions.

Consider the following greedy-type algorithm $\mathcal{G}$ for the MPP: $\mathcal{G}$ sorts the numbers such that $a_{\pi(1)} \geq a_{\pi(2)} \geq \cdots \geq a_{\pi(n)}$, initiates $X = \{\pi(1)\}$, $Y = \{\pi(2)\}$, and, for each $j \geq 3$, puts $\pi(j)$ into X if $\sum_{i \in X} a_i \leq \sum_{i \in Y} a_i$, and into Y, otherwise. It is easy to see that any solution X, Y produced by $\mathcal{G}$ satisfies $d(X, Y) \leq a_{\pi(1)}$.

Consider any solution X', Y' of the MPP for the input $\{a_1, a_2, \ldots, a_n\} - \{a_{\pi(1)}\}$. If we add $a_{\pi(1)}$ to Y' if $\sum_{i \in X'} a_i \leq \sum_{i \in Y'} a_i$ and to X', otherwise, then we obtain a

solution X'', Y'' for the original problem with $d(X'', Y'') \geq d(X, Y)$. Thus, the domination number of $\mathcal{G}$ is at least 2^{n-1} and we have the following:

Proposition 1.1 *The domination ratio of $\mathcal{G}$ is at least 0.5.*

In fact, a slight modification of $\mathcal{G}$ is of domination ratio very close to 1, see Section 3.

Let us consider another CO problem. In the *Assignment Problem* (AP), we are given a complete bipartite graph B with n vertices in each partite set and a non-negative weight weight wt (e) assigned to each edge e of B. We are required to find a perfect matching (i.e., a collection of n edges with no common vertices) in B of minimum total weight.

The AP can be solved to optimality in time $O(n^3)$ by the Hungarian algorithm. Thus, the domination number of the Hungarian algorithm equals $n!$, the total number of perfect matchings in B.

For some instances of the AP, the $O(n^3)$ time may be too high and thus we may be interested in having a faster heuristic for the AP. Perhaps, the first heuristics that comes into mind is the greedy algorithm (`greedy`). The greedy algorithm starts from the empty matching X and, at each iteration, it appends to X the cheapest edge of B that has no common vertices with edges already in X. (A description of `greedy` for a much more general combinatorial optimization problem is provided in Section 4.)

The proof of the following theorem shows that the greedy algorithm fails on many 'non-exotic' instances of the AP.

Theorem 1.2 *For the AP,* `greedy` *has domination number 1.*

Proof. Let B be a complete bipartite graph with n vertices in each partite set and let $u_1, u_2, \ldots, u_n$ and $v_1, v_2, \ldots, v_n$ be the two partite sets of B. Let M be any number greater than n. We assign weight $i \times M$ to the edge $u_i v_i$ for $i = 1, 2, \ldots, n$ and weight $\min\{i, j\} \times M + 1$ to every edge $u_i v_j$, $i \neq j$; see Figure 2 for illustration in the case $n = 3$.

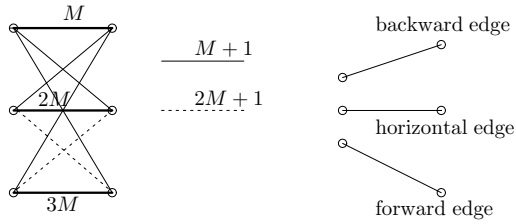


Figure 2. Assignment of weights for $n = 3$; classification of edges.

We classify edges of B as follows: $u_i v_j$ is horizontal (forward, backward) if $i = j$ ($i < j, i > j$). See Figure 2.

The greedy algorithm will choose edges $u_1v_1, u_2v_2, \ldots, u_nv_n$ (and in that order). We denote this perfect matching P and we will prove that P is the unique most expensive perfect matching in B. The weight of P is $(P) = M + 2M + \cdots + nM$.

Choose an arbitrary perfect matching P' in B distinct from P. Let P' have edges $u_1v_{p_1}, u_2v_{p_2}, \ldots, u_nv_{p_n}$. By the definition of the costs in B, wt $(u_i v_{p_i}) \leq M \times i + 1$. Since P' is distinct from P, it must have edges that are not horizontal. This means it has backward edges. If $u_k v_{p_k}$ is a backward edge, i.e. $p_k < k$, then wt $(u_k v_{p_k}) \leq M(k-1) + 1 = (Mk + 1) - M$. Hence,

$$\text{wt}(P') \leq (M + 2M + \cdots + nM) + n - M = \text{wt}(P) + n - M.$$

Thus, wt $(P') <$ wt(P). □

Question 1.3 *Formulate the greedy algorithm for the ATSP.*

Question 1.4 *Consider the following mapping f from the arc set of K_n^* into the edge set of $K_{n,n}$, the complete bipartite graph on $2n$ vertices. Let $x_1, \ldots, x_n$ be vertices of K_n^*, and let $\{u_1, \ldots, u_n\}$ and $\{v_1, \ldots, v_n\}$ be partite sets of $K_{n,n}$. Then $f(x_i x_j) = u_i v_{j-1}$ for each $1 \leq i \neq j \leq n$, where $v_0 = v_n$. Show that f maps every Hamilton cycle of K_n^* into a matching of $K_{n,n}$.*

Question 1.5 *Using the mapping f of Question 1.4 and Theorem 1.2, prove that the greedy algorithm has domination number 1 for the ATSP.*

1.3. Additional Terminology and Notation

Following the terminology in [20], a CO problem $\mathcal{P}$ is called $\mathcal{DOM}$*-easy* if there exists a polynomial time algorithm, $\mathcal{A}$, such that $domr(\mathcal{A}, n) \geq 1/p(n)$, where $p(n)$ is a polynomial in n. In other words, a problem is $\mathcal{DOM}$-easy, if, in polynomial time, we can always find a solution, with domination number at least a polynomial fraction of all solution. If no such algorithm exists, $\mathcal{P}$ is called $\mathcal{DOM}$*-hard*.

For a digraph D, $V(D)$ ($A(D)$) denotes the vertex (arc) set of H. The same notation are used for paths and cycles in digraphs. A *tour* in a digraph D is a Hamilton cycle in D. A *complete digraph* K_n^* is a digraph in which every pair x, y of distinct vertices is connected by the pair (x, y), (y, x) of arcs. The *out-degree* $d^+(v)$ (*in-degree* $d^-(v)$) of a vertex v of a digraph D is the number of arcs leaving v (entering v). It is clear that $|A(D)| = \sum_{v \in V(D)} d^+(v) = \sum_{v \in V(D)} d^-(v)$.

We will often consider weighted digraphs, i.e., pairs $(D,$ wt$)$, where wt is a mapping from $A(D)$ into the set of reals. For an arc $a = (x, y)$ in $(K_n^*,$ wt$)$, the *contraction* of

a results in a complete digraph with vertex set $V' = V(K_n^*) \cup \{v_a\} - \{x, y\}$ and weight function wt′, where $v_a \notin V(K^*_n)$, such that the weight wt′(u, w), for $u, w \in V'$, equals wt(u, x) if $w = v_a$, wt(y, w) if $u = v_a$, and wt(u, w), otherwise. The above definition has an obvious extension to a set of arcs; for more details, see [6].

For an undirected graph G, $V(G)$ $(E(G))$ denotes the vertex (edge) set of H. A *tour* in a graph G is a Hamilton cycle in G. A complete graph K_n is a graph in which every pair x, y of distinct vertices is connected by edge xy. Weighted graphs have weights assigned to their edges.

For a pair of given functions $f(k)$, $g(k)$ of a non-negative integer argument k, we say that $f(k) = O(g(k))$ if there exist positive constants c and k_0 such that $0 \leq f(k) \leq cg(k)$ for all $k \geq k_0$. If there exist positive constants c and k_0 such that $0 \leq cf(k) \leq g(k)$ for all $k \geq k_0$, we say that $g(k) = \Omega(f(k))$. Clearly, $f(k) = O(g(k))$ if and only if $g(k) = \Omega(f(k))$. If both $f(k) = O(g(k))$ and $f(k) = \Omega(g(k))$ hold, then we say that $f(k)$ and $g(k)$ are of the same order and denote it by $f(k) = \Theta(g(k))$.

2. TSP Heuristics with Large Domination Number

Since there is a recent survey on domination analysis of TSP heuristics [19], we restrict ourselves to giving a short overview of three important topics. All results will be formulated specifically for the ATSP or the STSP, but in many cases similar results hold for the symmetric or asymmetric counterparts as well.

2.1. ATSP Heuristics of Domination Number at Least $\Omega((n-2)!)$

We will show how the domination number of an ATSP heuristic can be related to the average value of a tour. This result was (up till now) used in all proofs that a heuristic has domination number at least $\Omega((n-2)!)$. Examples of such heuristics are the greedy expectation algorithm introduced in [21], vertex insertion algorithms and k-opt (see [19]). Using the above-mentioned result we will prove that vertex insertion algorithms have domination number at least $\Omega((n-2)!)$.

A *decomposition* of $A(K_n^*)$ into tours, is a collection of tours in K_n^*, such that every arc in K_n^* belongs to exactly one of the tours. The following lemma was proved for odd n by Kirkman (see [9], p. 187), and the even case result was established in [39].

Lemma 2.1 *For every $n \geq 2$, $n \neq 4$, $n \neq 6$, there exists a decomposition of $A(K_n^*)$ into tours.*

An *automorphism*, f, of $V(K_n^*)$ is a bijection from $V(K_n^*)$ to itself. Note that if C is a tour in K_n^* then $f(C)$ is also a tour K_n^*.

We define $\tau(K_n^*)$ as the average weight of a tour in K_n^*. As there are $(n-1)!$ tours in K_n^*, and $(n-2)!$ tours in K_n^*, which use a given arc e (see Question 2.2), we note that

$$\tau(K_n^*) = \frac{1}{(n-1)!} \sum_{e \in A(K_n^*)} \text{wt}(e) \times (n-2)!,$$

which implies that $\tau(K_n^*) = \text{wt}(K_n^*)/(n-1)$, where $\text{wt}(K_n^*)$ is the sum of all weights in K_n^*.

Question 2.2 *Let $e \in A(K_n^*)$ be arbitrary. Show that there are $(n-2)!$ tours in K_n^*, which use the arc e.*

Question 2.3 *Let $D = \{C_1, \ldots, C_{n-1}\}$ be a decomposition of $A(K_n^*)$ into tours. Assume that C_{n-1} is the tour in D of maximum weight. Show that $\text{wt}(C_{n-1}) \geq \tau(K_n^*)$.*

Question 2.4 *Let $D = \{C_1, \ldots, C_{n-1}\}$ be a decomposition of $A(K_n^*)$ into tours. Let α be an automorphism of $V(K_n^*)$. Prove that α maps D into a decomposition of $A(K_n^*)$ into tours.*

We are now ready to prove the main result of this section.

Theorem 2.5 *Assume that H is a tour in K_n^* such that $wt(H) \leq \tau(K_n^*)$. If $n \neq 6$, then H is not worse than at least $(n-2)!$ tours in K_n^*.*

Proof. The result is clearly true for $n = 2, 3$. If $n = 4$, the result follows from the fact that the most expensive tour, R, in K_n^* has weight at least $\tau(K_n^*) \geq \text{wt}(H)$. So the domination number of H is at least two (H and R are two tours of weight at least $\text{wt}(H)$).

Assume that $n \geq 5$ and $n \neq 6$. Let $V(K_n^*) = \{x_1, x_2, \ldots, x_n\}$. By Lemma 2.1 there exists a decomposition, $D = \{C_1, \ldots, C_{n-1}\}$ of $A(K_n^*)$ into tours. Furthermore there are $(n-1)!$ automorphisms, $\{\alpha_1, \alpha_2, \ldots, \alpha_{(n-1)!}\}$, of $V(K_n^*)$, which map vertex x_1 into x_1. Now let D_i be the decomposition of $A(K_n^*)$ into tours, obtained by using α_i on D. In other words, $D_i = \{\alpha_i(C_1), \alpha_i(C_2), \ldots, \alpha_i(C_{n-1})\}$ (see Question 2.4).

Note that if R is a tour in K_n^*, then R belongs to exactly $(n-1)$ decompositions in $\{D_1, D_2, \ldots, D_{(n-1)!}\}$, as one automorphism will map C_1 into R, another one will map C_2 into R, etc. Therefore R will lie in exactly the $(n-1)$ decompositions which we obtain from these $(n-1)$ automorphisms.

Now let E_i be the most expensive tour in D_i. By Question 2.3 we see that $\text{wt}(E_i) \geq \tau(K_n^*)$. As any tour in the set $\mathcal{E} = \{E_1, E_2, \ldots, E_{(n-1)!}\}$ appears at most $(n-1)$ times, the set $\mathcal{E}$ has at least $(n-2)!$ distinct tours, which all have weight at least $\tau(K_n^*)$. As $\text{wt}(H) \leq \tau(K_n^*)$, this proves the theorem. □

The above result has been applied to prove that a wide variety of ATSP heuristics have domination number at least $\Omega((n-2)!)$. Below we will show how the above result

can be used to prove that ATSP vertex insertion algorithms have domination number at least $(n-2)!$.

Let (K_n^*, wt) be an instance of the ATSP. Order the vertices $x_1, x_2, \ldots, x_n$ of K_n^* using some rule. The generic vertex insertion algorithm proceeds as follows. Start with the cycle $C_2 = x_1x_2x_1$. Construct the cycle C_j from C_{j-1} ($j = 3, 4, 5, \ldots, n$), by inserting the vertex x_j into C_{j-1} at the optimum place. This means that for each arc $e = xy$ which lies on the cycle C_{j-1} we compute $\text{wt}(xx_j) + \text{wt}(x_jy) - \text{wt}(xy)$, and insert x_j into the arc $e = xy$, which obtains the minimum such value. We note that $\text{wt}(C_j) = \text{wt}(C_{j-1}) + \text{wt}(xx_j) + \text{wt}(x_jy) - \text{wt}(xy)$.

Theorem 2.6 *The generic vertex insertion algorithm has domination number at least* $(n-2)!$.

Proof. We will prove that the generic vertex insertion algorithm produces a tour of weight at most $\tau(K_n^*)$ by induction. Clearly this is true for $n = 2$, as there is only one tour in this case. Now assume that it is true for K_{n-1}^*. This implies that $\text{wt}(C_{n-1}) \leq \tau(K_n^* - x_n)$. Without loss of generality assume that $C_{n-1} = x_1x_2 \ldots x_{n-1}x_1$. Let $\text{wt}(X, Y) = \sum_{x \in X,\ y \in Y} c(xy)$ for any disjoint sets X and Y. Since C_n was chosen optimally we see that its weight is at most (where $x_0 = x_{n-1}$ in the sum)

$$
\begin{aligned}
&\left(\sum_{i=0}^{n-2} \text{wt}(C_{n-1}) + \text{wt}(x_ix_n) + \text{wt}(x_nx_{i+1}) - \text{wt}(x_ix_{i+1})\right)/(n-1)\\
&\quad = \text{wt}(C_{n-1}) + (\text{wt}(V - x_n, x_n) + \text{wt}(x_n, V - x_n) - \text{wt}(C_{n-1}))/(n-1)\\
&\quad \leq ((n-2)\tau(K_n^* - x_n) + \text{wt}(V - x_n, x_n) + \text{wt}(x_n, V - x_n))/(n-1)\\
&\quad = (\text{wt}(K_n^* - x_n) + \text{wt}(V - x_n, x_n) + \text{wt}(x_n, V - x_n))/(n-1)\\
&\quad = \text{wt}(K_n^*)/(n-1) = \tau(K_n^*).
\end{aligned}
$$

This completes the induction proof. Theorem 2.5 now implies that the domination number of the generic vertex insertion algorithm is at least $(n-2)!$. □

2.2. Domination Numbers of Local Search Heuristics

In TSP local search (LS) heuristics, a neighborhood $N(T)$ is assigned to every tour T, a set of tours in some sense close to T. The *best improvement* LS proceeds as follows. We start from a tour T_0. In the i'th iteration ($i \geq 1$), we search in the neighborhood $N(T_{i-1})$ for the best tour T_i. If the weights of T_{i-1} and T_i do not coincide, we carry out the next iteration. Otherwise, we output T_i.

One of the first exponential size TSP neighborhoods (called `assign` in [12]) was considered independently by Sarvanov and Doroshko [38], and Gutin [17]. We describe this neighborhood and establish a simple upper bound on the domination number of the best improvement LS based on this neighborhood. We will see that the domination number of the best improvement LS based on `assign` is significantly smaller than that of the best improvement LS based on `2-opt`, a well-known STSP heuristic.

Consider a weighted K_n. Assume that $n = 2k$. Let $T = x_1y_1x_2y_2 \dots x_ky_kx_1$ be an arbitrary tour in K_n. The neighborhood `assign`, $N_a(T)$, is defined as follows: $N_a(T) = \{x_1y_{\pi(1)}x_2y_{\pi(2)} \dots x_ky_{\pi(k)}x_1 : (\pi(1), \pi(2), \dots, \pi(k))$ is a permutation of $(1, 2, \dots, k)\}$. Clearly, $N_a(T)$ contains $k!$ tours. We will show that we can find the tour of minimum weight in $N_a(T)$ in polynomial time.

Let B be a complete bipartite graph with partite sets $\{z_1, z_2, \dots, z_n\}$ and $\{y_1, y_2, \dots, y_n\}$, and let the weight of z_iy_j be $\mathrm{wt}(x_iy_j) + \mathrm{wt}(y_jx_{i+1})$ (where $x_{n+1} = x_1$). Let M be a perfect matching in B, and assume that z_i is matched to $y_{m(i)}$ in M. Observe that the weight of M is equal to the weight of the tour $x_1y_{m(1)}x_2y_{m(2)} \dots x_ny_{m(n)}x_1$. Since every tour in $N_a(T)$ corresponds to a perfect matching in B, and visa versa, a minimum weight perfect matching in B corresponds to a minimum weight tour in $N_a(T)$. Since we can find a minimum weight perfect matching in B in $O(n^3)$ time using the Hungarian method, we obtain the following theorem.

Theorem 2.7 *The best tour in $N_a(T)$ can be found in $O(n^3)$ time.*

While the size of $N_a(T)$ is quite large, the domination number of the best improvement LS based on `assign` is relatively small. Indeed, consider K_n with vertices $\{x_1, x_2, \dots, x_k, y_1, y_2, \dots, y_k\}$. Suppose that the weights of all edges of the forms x_iy_j and y_ix_j equal 1 and the weights of all other edges equal 0. Then, starting from the tour $T = x_1y_1x_2y_2 \dots x_ky_kx_1$ of weight n the best improvement will output a tour of weight n, too. However, there are only $(k!)^2/(2k)$ tours of weight n in K_n and the weight of no tour in K_n exceeds n. We have obtained the following:

Proposition 2.8 *For STSP, the domination number of the best improvement LS based on* `assign` *is at most $(k!)^2/(2k)$, where $k = n/2$.*

The k-`opt`, $k \geq 2$, neighborhood of a tour T consists of all tour that can be obtained by deleting a collection of k edges (arcs) and adding another collection of k edges (arcs). It is easy to see that one iteration of the best improvement k-`opt` LS can be completed in time $O(n^k)$. Rublineckii [36] showed that every local optimum for the best improvement 2-opt and 3-opt for STSP is of weight at least the average weight of a tour and, thus, by an analog of Theorem 2.5 is of domination number at least $(n-2)!/2$ when n is even and $(n-2)!$ when n is odd. Observe that this result is of restricted interest since to reach a k-opt local optimum one may need exponential time (cf. Section 3 in [27]). However, Punnen, Margot and Kabadi [34] managed to prove the following result.

Theorem 2.9 *For the STSP the best improvement* `2-opt` *LS produces a tour, which is not worse than at least $\Omega((n-2)!)$ other tours, in at most $O(n^3 \, logn)$ iterations.*

The last two assertions imply that after a polynomial number of iterations the best improvement `2-opt` LS has domination number at least $\Omega(2^n/n^{3.5})$ times larger than that of the best improvement `assign` LS.

Theorem 2.9 is also valid for the best improvement `3-opt` LS and some other LS heuristics for TSP, see [26, 34].

2.3. Upper Bounds for Domination Numbers of ATSP Heuristics

It is realistic to assume that any ATSP algorithm spends at least one unit of time on every arc of K_n^* that it considers. We use this assumption in the rest of this subsection.

Theorem 2.10 *[24, 22] Let $\mathcal{A}$ be an ATSP heuristic of complexity $t(n)$. Then the domination number of $\mathcal{A}$ does not exceed* $\max_{1\le n'\le n}(t(n)/n')^{n'}$.

Proof. Let $D = (K_n^*, \mathrm{wt})$ be an instance of the ATSP and let H be the tour that $\mathcal{A}$ returns, when its input is D. Let $DOM(H)$ denotes all tours in D which are not lighter than H including H itself. We assume that D is the worst instance for $\mathcal{A}$, namely $\mathrm{domn}(\mathcal{A}, n) = |DOM(H)|$. Since $\mathcal{A}$ is arbitrary, to prove this theorem, it suffices to show that $|DOM(H)| \le \max_{1\le n'\le n}(t(n)/n')^{n'}$.

Let E denote the set of arcs in D, which $\mathcal{A}$ actually examines; observe that $|E| \le t(n)$ by the assumption above. Let F be the set of arcs in H that are not examined by $\mathcal{A}$, and let G denote the set of arcs in $D - A(H)$ that are not examined by $\mathcal{A}$.

We first prove that every arc in F must belong to each tour of $DOM(H)$. Assume that there is a tour $H' \in DOM(H)$ that avoids an arc $a \in F$. If we assign to a a very large weight, H' becomes lighter than H, a contradiction.

Similarly, we prove that no arc in G can belong to a tour in $DOM(H)$. Assume that an $a \in G$ and a is in a tour $H' \in DOM(H)$. By making a very light, we can ensure that $\mathrm{wt}(H') < \mathrm{wt}(H)$, a contradiction.

Now let D' be the digraph obtained by contracting the arcs in F and deleting the arcs in G, and let n' be the number of vertices in D'. Note that every tour in $DOM(H)$ corresponds to a tour in D' and, thus, the number of tours in D' is an upper bound on $|DOM(H)|$. In a tour of D', there are at most $d^+(i)$ possibilities for the successor of a vertex i, where $d^+(i)$ is the out-degree of i in D'. Hence we obtain that

$$|DOM(H)| \le \prod_{i=1}^{n'} d^+(i) \le \left(\frac{1}{n'}\sum_{i=1}^{n'} d^+(i)\right)^{n'} \le \left(\frac{t(n)}{n'}\right)^{n'},$$

where we applied the arithmetic-geometric mean inequality. □

Corollary 2.11 *[24, 22] Let $\mathcal{A}$ be an ATSP heuristic of complexity $t(n)$. Then the domination number of $\mathcal{A}$ does not exceed* $\max\{e^{t(n)/e}, (t(n)/n)^n\}$*, where e is the basis of natural logarithms.*

Proof. Let $U(n) = \max_{1 \leq n' \leq n}(t(n)/n')^{n'}$. By differentiating $f(n') = (t(n)/n')^{n'}$ with respect to n' we can readily obtain that $f(n')$ increases for $1 \leq n' \leq t(n)/e$, and decreases for $t(n)/e \leq n' \leq n$. Thus, if $n \leq t(n)/e$, then $f(n')$ increases for every value of $n' < n$ and $U(n) = f(n) = (t(n)/n)^n$. On the other hand, if $n \geq t(n)/e$ then the maximum of $f(n')$ is for $n' = t(n)/e$ and, hence, $U(n) = e^{t(n)/e}$. □

The next assertion follows directly from the proof of Corollary 2.11.

Corollary 2.12 *[24, 22] Let $\mathcal{A}$ be an ATSP heuristic of complexity $t(n)$. For $t(n) \geq en$, the domination number of $\mathcal{A}$ does not exceed $(t(n)/n)^n$.*

Note that the restriction $t(n) \geq en$ is important since otherwise the bound of Corollary 2.12 can be invalid. Indeed, if $t(n)$ is a constant, then for n large enough the upper bound becomes smaller than 1, which is not correct since the domination number is always at least 1.

Question 2.13 *Fill in details in the proof of Corollary 2.11.*

Question 2.14 *Using Corollary 2.11 show that ATSP $O(n)$-time algorithms can have domination number at most $2^{\Theta(n)}$.*

Question 2.15 *Show that there exist ATSP $O(n)$-time algorithms of domination number at least $2^{\Omega(n)}$. Compare the results of the last two questions.*

We finish this section with a result from [24] that improves (and somewhat clarifies) Theorem 20 in [34]. The proof is a modification of the proof of Theorem 20 in [34].

Theorem 2.16 *Unless $P = NP$, there is no polynomial time ATSP algorithm of domination number at least $(n-1)! - \lfloor n - n^\alpha \rfloor !$ for any constant $\alpha < 1$.*

Proof. Assume that there is a polynomial time algorithm $\mathcal{H}$ with domination number at least $(n-1)! - \lfloor n - n^\alpha \rfloor !$ for some constant $\alpha < 1$. Choose an integer $s > 1$ such that $\frac{1}{s} < \alpha$.

Consider a weighted complete digraph (K_n^*, w). We may assume that all weights are non-negative as otherwise we may add a large number to each weight. Choose a pair u, v of vertices in K_n^*. Add, to K_n^*, another complete digraph D on $n^s - n$ vertices, in which all weights are 0. Append all possible arcs between K_n^* and D such that the weights of all arcs coming into u and going out of v are 0 and the weights of all other arcs are M, where M is larger than n times the maximum weight in (K_n^*, w). We have obtained an instance $(K_{n^s}^*, w')$ of ATSP.

Apply $\mathcal{H}$ to $(K_{n^s}^*, w')$ (observe that $\mathcal{H}$ is polynomial for $(K_{n^s}^*, w')$). Notice that there are exactly $(n^s - n)!$ Hamilton cycles in $(K_{n^s}^*, w')$ of weight L, where L is the weight of a lightest Hamilton (u, v)-path in K_n^*. Each of the $(n^s - n)!$ Hamilton cycles is obviously

optimal. Observe that the domination number of $\mathcal{H}$ is at least $(n^s-1)! - \lfloor n^s - (n^s)^\alpha \rfloor !$. However, for sufficiently large n, we have

$$(n^s-1)! - \lfloor n^s - (n^s)^\alpha \rfloor ! \geq (n^s-1)! - (n^s-n)! + 1$$

as $n^{s\alpha} \geq n+1$ for n large enough. Thus, a Hamilton cycle produced by $\mathcal{H}$ is always among the optimal solutions (for n large enough). This means that we can obtain a lightest Hamilton (u, v)-path in K_n^* in polynomial time, which is impossible since the lightest Hamilton (u, v)-path problem is a well-known NP-hard problem. We have arrived at a contradiction. □

For a result that is stronger than Theorem 2.16, see [34].

3. Heuristics of Large Domination Ratio for Other CO Problems

In this section, we consider other CO problems which have heuristics with relatively large domination ratios, as well as some CO problems which provably do not have heuristics with large domination ratios (unless P = NP). Even though most initial research on domination analysis has been done on TSP, there is now a wide variety of other problems, which have been studied in this respect.

3.1. Minimum Partition and Multiprocessor Scheduling

We already considered the Minimum Partition Problem (MPP) in Subsection 1.2, where we described a simple algorithm of domination ratio at least 0.5. In this subsection we introduce a slightly more complicated algorithm of domination ratio close to 1.

Let B_n be the set of all n-dimensional vectors $(\epsilon_1, \epsilon_2, \ldots, \epsilon_n)$ with $\{-1, 1\}$ coordinates. The MPP can be stated as follows: given n nonnegative numbers $\{a_1, a_2, \ldots, a_n\}$, find a vector $(\epsilon_1, \epsilon_2, \ldots, \epsilon_n) \in B_n$ such that $|\sum_{i=1}^n \epsilon_i a_i|$ is minimum.

Consider the following greedy-type algorithm $\mathcal{B}$. Initially sort the numbers such that $a_{\pi(1)} \geq a_{\pi(2)} \geq \cdots \geq a_{\pi(n)}$. Choose an integral constant $p > 0$ and fix $k = \lfloor p \log_2 n \rfloor$. Solve the MP to optimality for $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(k)}$, i.e., find optimal values of $\epsilon_{\pi(1)}, \epsilon_{\pi(2)}, \ldots, \epsilon_{\pi(k)}$. (This can be trivially done in time $O(n^p)$.) Now for each $j > k$, if $\sum_{i=1}^{j-1} \epsilon_{\pi(i)} a_{\pi(i)} < 0$, then set $\epsilon_{\pi(j)} = +1$, and otherwise $\epsilon_{\pi(j)} = -1$.

Theorem 3.1 *[2] The domination ratio of $\mathcal{B}$ is at least* $1 - \binom{k}{\lfloor k/2 \rfloor}/2^k = 1 - \Theta(\frac{1}{\sqrt{k}})$.

To prove this theorem, without loss of generality, we may assume $a_1 \geq a_2 \geq \cdots \geq a_n$.

Observe that if $\min |\sum_{i=1}^k \epsilon_i a_i| \geq \sum_{i=k+1}^n a_i$, then $\mathcal{B}$ outputs an optimal solution. Otherwise, it can be easily proved by induction that the solution produced satisfies $|\sum_{i=1}^n \epsilon_i a_i| \leq a_{k+1}$. Thus, we may assume the last inequality.

Now it suffices to prove the following:

Proposition 3.2 *The number of vectors* $(\epsilon_1, \ldots, \epsilon_n) \in B_n$ *for which* $|\sum_{i=1}^{n} \epsilon_i a_i| < a_{k+1}$ *is at most* $\binom{k}{\lfloor k/2 \rfloor} 2^{n-k}$.

To prove this proposition, we will use the following lemma:

Lemma 3.3 *[13] Let* $a_1 \geq a_2 \geq \cdots \geq a_k$ *and let* (a, b) *be an arbitrary open interval such that* $b - a \leq 2a_k$. *Then the number of vectors* $(\delta_1, \ldots, \delta_k) \in B_k$ *such that* $\sum_{i=1}^{k} \delta_i a_i \in (a, b)$ *is at most* $\binom{k}{\lfloor k/2 \rfloor}$.

Fix a vector $(\epsilon_{k+1}, \ldots, \epsilon_n) \in B_{n-k}$. Denote the sum $\sum_{i=k+1}^{n} \epsilon_i a_i$ by S. Now $|\sum_{i=1}^{n} \epsilon_i a_i| < a_{k+1}$ if and only if $\sum_{i=1}^{k} \epsilon_i a_i$ belongs to the open interval $(-S - a_{k+1}, -S + a_{k+1})$. However, by the lemma above, there are at most $\binom{k}{\lfloor k/2 \rfloor}$ vectors $(\epsilon_1, \ldots, \epsilon_k)$ with this property. Since we can fix $(\epsilon_{k+1}, \ldots, \epsilon_n) \in B_{n-k}$ in $|B_{n-k}| = 2^{n-k}$ ways, the assertion of the proposition follows, implying the assertion of Theorem 3.1 as well.

For an integer $p \geq 2$, a p-partition of a set A is a collection $A_1, A_2, \ldots, A_p$ of subsets of A such that $\cup_{i=1}^{p} A_i = A$ and $A_i \cap A_j = \emptyset$ for each $1 \leq i \neq j \leq p$. Theorem 3.1 was generalized in [18], where the following minimum p-processor scheduling problem was considered. We are given an integer $p \geq 2$ and a sequence $w_1, w_2, \ldots, w_n$ of positive integers, and we are required to find a p-partition $N_1, N_2, \ldots, N_p$ of $\{1, 2, \ldots, n\}$ such that $\max_{i=1}^{p} \sum_{j \in N_i} w_j$ is as small as possible. Notice that the minimum 2-processor scheduling problem is equivalent to MMP from the Domination Analysis point of view.

3.2. Max Cut

The Max Cut (MC) is the following problem: given a weighted complete graph $G = (V, E, \text{wt})$, find a bipartition (a *cut*) (X, Y) of V such that the sum of weights of the edges with one end vertex in X and the other in Y, called the *weight of the cut* (X, Y), is maximum.

We will show that the MC is $\mathcal{DOM}$-easy, just as TSP is. (For the definition of $\mathcal{DOM}$-easy problems, see Subsection 1.3.)

Theorem 3.4 *[20] The MC is* $\mathcal{DOM}$*-easy. In fact, there is an algorithm, for the MC, of domination number at least* $\Omega(2^n/n)$.

Proof. Let $G = (V, E)$ be a weighted complete graph with $n = |V|$ vertices and let W be the sum of the weights of the edges in G. Clearly, the average weight of a cut of G is $\overline{W} = W/2$.

Consider the following well-known approximation algorithm $\mathcal{C}$ that always produces a cut of weight at least $\overline{W}$. The algorithm $\mathcal{C}$ considers the vertices of G in any fixed order $v_1, v_2, \ldots, v_n$, initiates $X = \{v_1\}$, $Y = \{v_2\}$, and adds v_i, $i \geq 3$, to X or Y

depending on whether the sum of the weights of edges between v and Y or between v and X is larger. We will prove that $\mathcal{C}$ is of domination number at least $\Omega(2^n/n)$. To show this, it suffices to prove that the cuts in G of weight at most $\overline{W}$ (we call them *bad cuts*) constitute at least an $O(1/n)$ part of all cuts.

We call a cut (X, Y) of G a *k-cut* if $|X| = k$. We evaluate the fraction of bad cuts among k-cuts when $k \leq n/2 - 2\sqrt{n}$.

For a fixed edge uv of G among $\binom{n}{k}$ k-cuts there are $2\binom{n-2}{k-1}$ k-cuts that contain uv. Thus, the average weight of a k-cut is $\overline{W}_k = 2\binom{n-2}{k-1}W/\binom{n}{k}$. Let b_k be the number of bad k-cuts. Then, $(\binom{n}{k} - b_k)\overline{W}/\binom{n}{k} \leq \overline{W}_k$. Hence,

$$b_k \geq \binom{n}{k} - 4\binom{n-2}{k-1} \geq \binom{n}{k}\left(1 - \frac{4k(n-k)}{n(n-1)}\right).$$

It is easy to verify that $1 - 4k(n-k)/(n(n-1)) > 1/n$ for all $k \leq n/2 - 2\sqrt{n}$. Hence, G has more than $\frac{1}{n}\sum_{k \leq n/2-2\sqrt{n}} \binom{n}{k}$ bad cuts. By the famous DeMoivre-Laplace theorem of probability theory, it follows that the last sum is at least $c2^n$ for some positive constant c. Thus, G has more than $c2^n/n$ bad cuts. □

Using a more advanced probabilistic approach Alon, Gutin and Krivelevich [2] recently proved that the algorithm $\mathcal{C}$ described above is of domination ratio larger than 0.025.

3.3. Max-k-SAT

One of the best-known NP-complete decision problems is 3-SAT. This problem is the following: We are given a set V of variables and a collection C of clauses each with exactly 3 literals (a literal is a variable or a negated variable in V; in a clause, literals are separated by "OR"'s). Does there exist a truth assignment for V, such that every clause is *true*?

We will now consider the more general optimization problem max-k-SAT. This is similar to 3-SAT, but there are k literals in each clause, and we want to find a truth assignment for V which maximizes the number of clauses that are *true*, i.e., *satisfied*. Let $U = \{x_1, \ldots, x_n\}$ be the set of variables in the instance of max-k-SAT under consideration. Let $\{C_1, \ldots, C_m\}$ be the set of clauses. We assume that k is a constant.

Berend and Skiena [10] considered some well-known algorithms for max-k-SAT and the algorithms turned out to have domination number at most $n + 1$. However an algorithm considered in [20] is of domination number at least $\Omega(2^n/n^{\lfloor k/2 \rfloor})$. We will study this algorithm.

Assign a truth assignment to all the variables at random. Let p_i be the probability that C_i (i'th clause) is satisfied. Observe that if some variable and its negation belong

to C_i, then $p_i = 1$, otherwise $p_i = 1 - 2^{-k'}$ where k' is the number of distinct variables in C_i. Thus, the average number of satisfied clauses in a random truth assignment is $E = \sum_{i=1}^{m} p_i$.

For simplicity, in the sequel *true* (*false*) will be replaced by the binaries 1 (0). By a construction described in Section 15.2 of [3], there exists a binary matrix $A = (a_{ij})$ with n columns and $r = O(n^{\lfloor k/2 \rfloor})$ rows such that the following holds: Let B be an arbitrary submatrix of A, consisting of k of its columns (chosen arbitrarily), and all r of its rows. Every binary k-vector coincides with exactly $r/2^k$ rows of B. We give a short example below, with $n = 4$ and $k = 3$ ($r = 8$). The matrix A can be constructed in polynomial time [3].

0	0	0	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
1	1	1	1

Note that no matter which 3 columns we consider, we will always get the vectors (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1) equally many times (in this case, once) in the 8 rows.

Observe that each row, say row j, corresponds to a truth assignment β_j (where the i'th variable gets the truth assignment of the i'th column, i.e. $x_1 = a_{j1}, \ldots, x_n = a_{jn}$). Let T_j be the number of clauses satisfied by β_j. Consider a polynomial time algorithm $\mathcal{S}$ that computes $T_1, \ldots, T_r$ and outputs $\beta^*(A)$ that satisfies $T^*(A) = \max_{j=1}^{r} T_j$ clauses.

We will prove that $\mathcal{S}$ has domination number at least $\Omega(2^n/n^{\lfloor k/2 \rfloor})$. Since rp_i of the r truth assignments will satisfy the i'th clause, we conclude that $\sum_{i=1}^{r} T_i = rE$ (recall that $E = \sum_{i=1}^{m} p_i$; see also Question 3.6). Therefore the truth assignment $\beta^*(A)$ must satisfy at least E clauses. Furthermore by a similar argument we conclude that the row $\beta_*(A)$ corresponding to the truth assignment with fewest satisfied clauses, which we shall call $W(A)$, has at most E satisfied clauses.

Let $X \subseteq \{1, 2, \ldots, n\}$ be arbitrary and let A_X be the matrix obtained from A by changing all 0's to 1's and all 1's to 0's in the i'th column in A, for all $i \in X$. Note that A_X has the same properties as A. Observe that a truth assignment can appear at most r times in $\mathcal{T} = \{\beta_*(A_X) : X \subseteq \{1, 2, \ldots, n\}\}$, as a truth assignment cannot appear in the j'th row of A_X and A_Y, if $X \neq Y$. Therefore $\mathcal{T}$ contains at least $2^n/r$ distinct truth assignments all with at most E satisfied clauses. Therefore, we have proved the following:

Theorem 3.5 *[20] The algorithm $\mathcal{S}$ is of domination number at least $\Omega(2^n/n^{\lfloor k/2 \rfloor})$.*

Question 3.6 *Consider the given algorithm for max-k-SAT. Prove that rp_i rows will result in the i'th clause being* true*, so $\sum_{i=1}^{r} T_i = rE$.*

Question 3.7 *[20] Show that Theorem 3.5 can be extended to the weighted version of max-k-SAT, where each clause C_i has a weight w_i and we wish to maximize the total weight of satisfied clauses.*

Alon, Gutin and Krivelevich [2] recently proved, using an involved probabilistic argument, that the algorithm of Theorem 3.5 is, in fact, of domination number $\Omega(2^n)$.

3.4. Fixed Span Frequency Assignment Problem

In [31] the domination number is computed for various heuristics for the *Fixed Span Frequency Assignment Problem* (fs-FAP), which is defined as follows. We are given a set of vertices $\{x_1, x_2, \ldots, x_n\}$ and an $n \times n$ matrix $C = (c_{ij})$. We want to assign a frequency f_i to each vertex x_i, such that $|f_i - f_j| \geq c_{ij}$ for all $i \neq j$. However when f_i has to be chosen from a set of frequencies $\{0, 1, \ldots, \sigma - 1\}$, where σ is a fixed integer, then this is not always possible. If $|f_i - f_j| < c_{ij}$, then let $x_{ij} = 1$, and otherwise let $x_{ij} = 0$.

We are also given a matrix $W = (w_{ij})$ of weights, and we want to minimize the sum $\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} w_{ij}$. In other words we want to minimize the weight of all the edges that are *broken* (i.e. which have $|f_i - f_j| < c_{ij}$). Put $c_{ii} = 0$ for all $i = 1, 2, \ldots, n$. Since every vertex may be assigned a frequency in $\{0, 1, \ldots, \sigma - 1\}$, the following holds.

Proposition 3.8 *The total number of solutions for the fs-FAP is σ^n.*

A heuristic for the fs-FAP, which has similarities with the greedy expectation algorithm for the TSP (see [21]) is as follows (see [31] for details). We will assign a frequency to each vertex $x_1, x_2, \ldots, x_n$, in that order. Assume that we have already assigned frequencies to $x_1, x_2, \ldots, x_{i-1}$ and suppose that we assign frequency f_i to x_i. For all $j > i$, let p_{ij} be the probability that $|f_i - f_j| < c_{ij}$, if we assign a random frequency to j. For all $j < i$ let $x_{ij} = 1$ if $|f_i - f_j| < c_{ij}$ and $x_{ij} = 0$ otherwise. We now assign the frequency f_i to x_i, which minimizes the following:

$$\sum_{j=1}^{i-1} w_{ij} x_{ij} + \sum_{j=i+1}^{n} w_{ij} p_{ij}.$$

In other words we choose the frequency which minimizes the weight of the constraints that get broken added to the average weight of constraints that will be broken by assigning the remaining vertices with random frequencies. It is not too difficult to see that the above approach produces an assignment of frequencies, such that the weight of the broken edges is less than or equal to the average, taken over all assignments of frequencies. Koller and Noble [31] proved the following theorem where $\mathcal{G}$ is the algorithm described above.

Theorem 3.9 *[31] The domination number of $\mathcal{G}$ is at least $\sigma^{n-\lceil log_2 n \rceil - 1}$.*

Note that the following holds.

$$\sigma^{n-\lceil log_2 n\rceil-1} \geq \sigma^{n-log_2 n-2} \geq \frac{\sigma^n}{\sigma^2 n^{log_2 \sigma}}$$

Therefore $\mathcal{G}$ finds a solution which is at least as good as a polynomial fraction of all solutions (σ is a constant). This means that the fs-FAP is $\mathcal{DOM}$-easy.

In [31] it is furthermore shown that $\mathcal{G}$ has higher domination number than the better-known greedy-type algorithm, which minimizes only $\sum_{j=1}^{i-1} w_{ij}x_{ij}$, in each step.

3.5. $\mathcal{DOM}$-hard Problems

In this section we consider two well-known graph theory problems, which are somewhat different from the previous problems we have considered. Firstly, the number of feasible solutions, for an input of size n, depends on the actual input, and not just its size.

A *clique* in a graph G is a set of vertices in G such that every pair of vertices in the set are connected by an edge. The *Maximum Clique Problem* (MCl) is the problem of finding a clique of maximum cardinality in a graph. A *vertex cover* in a graph G is a set S of vertices in G such that every edge is incident to a vertex in S. The *Minimum Vertex Cover Problem* (MVC) is the problem of finding a minimum cardinality vertex cover. It is easy to see that the number of cliques in a graph depends on its structure, and not only on the number of vertices. The same holds for vertex covers.

The problems we have considered in the previous subsections have been $\mathcal{DOM}$-easy. We will show that MCl and MVC are $\mathcal{DOM}$-hard unless P = NP.

Theorem 3.10 *[20] MCl is $\mathcal{DOM}$-hard unless P = NP.*

Proof. We use a result by Håstad [29], which states that, provided that P $\neq$ NP, MCl is not approximable within a factor $n^{1/2-\epsilon}$ for any $\epsilon > 0$, where n is the number of vertices in a graph.

Let G be a graph with n vertices, and let q be the number of vertices in a maximum clique Q of G. Let $\mathcal{A}$ be a polynomial time algorithm and let $\mathcal{A}$ find a clique M with m vertices in G.

Since the clique Q 'dominates' all 2^q of its subcliques and the clique M 'dominates' at most $\binom{n}{m}2^m$ cliques in G, the domination ratio r of $\mathcal{A}$ is at most $\binom{n}{m}2^m/2^q$. By the above non-approximability result of Håstad [29], we may assume that $mn^{0.4} \leq q$. Thus,

$$r \leq \frac{\binom{n}{m}2^m}{2^q} \leq \frac{(en/m)^m 2^m}{2^q} \leq \frac{(n/m)^m(2e)^m}{2^{mn^{0.4}}} = 2^s,$$

where $s = m(\log n - \log m + 1 + \log e - n^{0.4})$. Clearly, 2^s is smaller than $1/p(n)$ for any polynomial $p(n)$ when n is sufficiently large. □

An *independent set* in a graph is a set S of vertices such that no edge is joins two vertices in S. The *Maximum Independent Set* problem (MIS) is the problem of finding a minimum cardinality independent set in a graph.

Question 3.11 *Using Theorem 3.10 prove that MIS is $\mathcal{DOM}$-hard unless P=NP.*

Question 3.12 *Let $G = (V, E)$ be a graph. Prove that S in an independent set in G if and only if $V - S$ is a vertex cover in G.*

Question 3.13 *Using the results of the previous two questions prove that MVC is $\mathcal{DOM}$-hard unless P = NP.*

3.6. Other Problems

There are many other combinatorial optimization problems studied in the literature that were not considered above. We will overview some of them.

In the *Generalized TSP*, we are given a weighted complete directed or undirected graph G and a partition of its vertices into non-empty sets $V_1, \ldots, V_k$. We are required to compute a lightest cycle in G containg exactly one vertex from each $V_i, i = 1, \ldots, k$. In the case when all V_i's are of the same cardinality, Ben-Arieh et al. [8] proved that the Generalized TSP is $\mathcal{DOM}$-easy.

The *Quadratic Assignment Problem* (QAP) can be formulated as follows. We are given two $n \times n$ matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ of integers. Our aim is to find a permutation π of $\{1, 2, \ldots, n\}$ that minimizes the sum

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)}.$$

Using group-theoretical approaches, Gutin and Yeo [25] proved only that QAP is $\mathcal{DOM}$-easy when n is a prime power.

Conjecture 3.14 *QAP is $\mathcal{DOM}$-easy (for every value of n).*

It was noted in [20] that Theorem 3.10 holds for some cases of the following general problem: the *Maximum Induced Subgraph with Property* Π (MISP), see Problem GT25 in the compendium of [4]). The property Π must be hereditary, i.e., every induced subgraph of a graph with property Π has property Π, and non-trivial, i.e., it is satisfied for infinitely many graphs and false for infinitely many graphs. Lund and Yannakakis [32] proved that MISP is not approximable within n^{ϵ} for some $\epsilon > 0$ unless P = NP, if

Π is hereditary, non-trivial and is false for some clique or independent set (e.g., planar, bipartite, triangle-free). This non-approximability result can be used as in the proof of Theorem 3.10.

4. Greedy Algorithm

The main practical message of this section is that one should be careful while using the classical greedy algorithm in combinatorial optimization (CO): there are many instances of CO problems for which the greedy algorithm will produce the unique worst possible solution. Moreover, this is true for several well-known optimization problems and the corresponding instances are not exotic, in a sense. This means that not always the paradigm of greedy optimization provides any meaningful optimization at all.

In this section we provide a wide extension of Theorem 1.2, which slightly generalizes the main theorem in [23]. Interestingly, the proof of the extension is relatively easy.

An *independence system* is a pair consisting of a finite set E and a family $\mathcal{F}$ of subsets (called *independent sets*) of E such that (I1) and (I2) are satisfied.

(I1) The empty set is in $\mathcal{F}$;
(I2) If $X \in \mathcal{F}$ and Y is a subset of X, then $Y \in \mathcal{F}$.

A maximal (with respect to inclusion) set of $\mathcal{F}$ is called a *base*. Clearly, an independence system on a set E can be defined by its bases. Notice that bases may be of different cardinality.

Many combinatorial optimization problems can be formulated as follows. We are given an independence system $(E, \mathcal{F})$ and a weight function wt that assigns an integral weight $\mathrm{wt}(e)$ to every element $e \in E$. The weight $\mathrm{wt}(S)$ of $S \in \mathcal{F}$ is defined as the sum of the weights of the elements of S. It is required to find a base $B \in \mathcal{F}$ of minimum weight. In this section, we will consider only such problems and call them the $(E, \mathcal{F})$-*optimization problems*.

If $S \in \mathcal{F}$, then let $I(S) = \{x : \ S \cup \{x\} \in \mathcal{F}\} - S$. The greedy algorithm (or, `greedy`, for short) constructs a base as follows: `greedy` starts from an empty set X, and at every step `greedy` takes the current set X and adds to it a minimum weight element $e \in I(X)$; `greedy` stops when a base is built.

Consider the following example. Let $E' = \{a, b, c, d\}$. We define an independence system $(E', \mathcal{F}')$ by listing its two bases: $\{a, b, c\}$, $\{c, d\}$. Recall that the independent sets of $(E', \mathcal{F}')$ are the subsets of its bases. Let the weights of a, b, c, d be $1, 5, 0, 2$, respectively. (Notice that the weight assignment determines an instance of the $(E', \mathcal{F}')$-optimization problem.) `greedy` starts from $X = \emptyset$, then adds c to X. At the next iteration it appends a to X. `greedy` cannot add d to $X = \{a, c\}$ since $d \notin I(X)$. Thus, `greedy` appends b to X and stops.

Since $6 = \mathrm{wt}(a, b, c) > \mathrm{wt}(c, d) = 2$, the domination number of `greedy` is 1 for this instance of the $(E', \mathcal{F}')$-optimization problem.

Note that if we add (I3) below to (I1),(I2), then we obtain one of the definitions of a matroid [33]:

(I3) If U and V are in $\mathcal{F}$ and $|U| > |V|$, then there exists $x \in U - V$ such that $V \cup \{x\} \in \mathcal{F}$.

It is well-known that domination number of `greedy` for every matroid $(E, \mathcal{F})$ is $|\mathcal{F}|$: `greedy` always finds an optimum for the $(E, \mathcal{F})$-optimization problem. Thus, it is surprising to have the following theorem that generalizes Theorem 1.2.

Theorem 4.1 *[23, 24] Let $(E, \mathcal{F})$ be an independence system and $B' = \{x_1, \ldots, x_k\}$, $k \geq 2$, a base. Suppose that the following holds for every base $B \in \mathcal{F}$, $B \neq B'$,*

$$\sum_{j=0}^{k-1} |I(x_1, x_2, \ldots, x_j) \cap B| < k(k+1)/2. \tag{1}$$

Then the domination number of `greedy` *for the $(E, \mathcal{F})$-optimization problem equals 1.*

Proof. Let M be an integer larger than the maximal cardinality of a base in $(E, \mathcal{F})$. Let $\mathrm{wt}(x_i) = iM$ and $\mathrm{wt}(x) = 1 + jM$ if $x \notin B'$, $x \in I(x_1, x_2, \ldots, x_{j-1})$ but $x \notin I(x_1, x_2, \ldots, x_j)$. Clearly, `greedy` constructs B' and $\mathrm{wt}(B') = Mk(k+1)/2$.

Let $B = \{y_1, y_2, \ldots, y_s\}$ be a base different from B'. By the choice of wt made above, we have that $\mathrm{wt}(y_i) \in \{aM, aM + 1\}$ for some positive integer a.

Clearly

$$y_i \in I(x_1, x_2, \ldots, x_{a-1}),$$

but $y_i \notin I(x_1, x_2, \ldots, x_a)$. Hence, by (I2), y_i lies in $I(x_1, x_2, \ldots, x_j) \cap B$, provided $j \leq a - 1$. Thus, y_i is counted a times in $\sum_{j=0}^{k-1} |I(x_1, x_2, \ldots, x_j) \cap B|$. Hence,

$$\begin{aligned} \mathrm{wt}(B) = \sum_{i=1}^{s} \mathrm{wt}(y_i) &\leq s + M \sum_{j=0}^{k-1} |I(x_1, x_2, \ldots, x_j) \cap B| \\ &\leq s + M(k(k+1)/2 - 1) = s - M + \mathrm{wt}(B'), \end{aligned}$$

which is less than the weight of B' as $M > s$. Since $\mathcal{A}$ finds B', and B is arbitrary, we see that `greedy` finds the unique heaviest base. □

The strict inequality (1) cannot be relaxed to the non-strict one due to Question 4.3.

Question 4.2 *Let $(E, \mathcal{F})$ be a matroid. Using (I3), show that for two distinct bases B and $B' = \{x_1, x_2, \ldots, x_k\}$, we have that $|I(x_1, x_2, \ldots, x_j) \cap B| \geq k - j$ for $j = 0, 1, \ldots, k$. Thus,*

$$\sum_{j=0}^{k-1} |I(x_1, x_2, \ldots, x_j) \cap B| \geq k(k+1)/2.$$

Question 4.3 *[23] Consider a matroid $(E', \mathcal{F}')$ in which E' consists of the columns of matrix $M = (I|2I)$, where I is the $k \times k$ identity matrix, and $\mathcal{F}'$ consists of collections of linearly independent columns of M. (Check that $(E', \mathcal{F}')$ is a matroid.) Let B and $B' = \{x_1, x_2, \ldots, x_k\}$ be two distinct bases of our matroid. Show that $\sum_{j=0}^{k-1} |I(x_1, x_2, \ldots, x_j) \cap B| = k(k+1)/2$.*

Recall that by the Assignment Problem (AP) we understand the problem of finding a lightest perfect matching in a weighted complete bipartite graph $K_{n,n}$.

Question 4.4 *Prove Corollary 4.5 applying Theorem 4.1*

Corollary 4.5 *[23] Every greedy-type algorithm $\mathcal{A}$ is of domination number 1 for the Asymmetric TSP, Symmetric TSP and AP.*

Bang-Jensen, Gutin and Yeo [7] considered the $(E, \mathcal{F})$-optimization problems, in which wt assumes only a finite number of integral values. For such problems, the authors of [7] completely characterized all cases when `greedy` *may* construct the unique worst possible solution. Here the word may means that `greedy` may choose any element of E of the same weight. Their characterization can be transformed into a characterization for the case when wt assumes any integral values. An *ordered partitioning* of an ordered set $Z = \{z_1, z_2, \ldots, z_k\}$ is a collection of subsets $A_1, A_2, \ldots, A_q$ of Z satisfying that if $z_r \in A_i$ and $z_s \in A_j$ where $1 \leq i < j \leq q$ then $r < s$. Some of the sets A_i may be empty and $\cup_{i=1}^{q} A_i = Z$.

Theorem 4.6 *Let $(E, \mathcal{F})$ be independence system. There is a weight assignment w such that the greedy algorithm may produce the unique worst possible base if and only if $\mathcal{F}$ contains some base B with the property that for some ordering $x_1, \ldots, x_k$ of the elements of B and some ordered partitioning $A_1, A_2, \ldots, A_r$ of $x_1, \ldots, x_k$ the following holds for every base $B' \neq B$ of $\mathcal{F}$:*

$$\sum_{j=0}^{r-1} |I(A_{0,j}) \cap B'| < \sum_{j=1}^{r} j \times |A_j|, \tag{2}$$

where $A_{0,j} = A_0 \cup \ldots \cup A_j$ and $A_0 = \emptyset$.

5. Practicality of Domination Analysis

Earlier in this chapter we have seen that domination analysis (DA) provides theoretical explanations of the poor computational behavior of `greedy` for certain optimization problems and of the fact that some very large neighborhoods in local search are computationally much weaker than some 'small' neighborhoods.

One might wonder whether a heuristic $\mathcal{A}$, that is significantly better that another heuristic $\mathcal{B}$ from the point of view of DA, is better that $\mathcal{B}$ in computational experiments. In particular, whether `greedy` is worse, in computational experiments, than any ATSP heuristic of domination number at least $(n-2)!$? Generally speaking the answer to this natural question is negative. This is because computational experiments and domination analysis indicate different aspects of quality of heuristics. Nevertheless, it seems that many heuristics of very small domination number such as `greedy` for TSP fail also in computational experiments and thus are not very robust.

Koller and Noble [31] showed that the heuristic $\mathcal{G}$ that they introduced for the frequency assignment problem is of larger domination number than the well-known greedy algorithm. However, the greedy algorithm is usually better in computational experiments than $\mathcal{G}$. Judging only by the computational experiments, $\mathcal{G}$ is of no interest. However, $\mathcal{G}$ might well be of interest when difficult instances of the frequency assignment problem are considered.

Ben-Arieh et al. [8] studied some heuristics for the Generalized TSP defined above. They investigated three modifications of a generic heuristic. In the computational experiment in [8] one of the modifications was clearly inferior to the other two. The best two behaved very similarly. Nevertheless, the authors of [8] managed to 'separate' the two modifications by showing that one of the modifications was of much larger domination number.

Acknowledgment

We would like to thank Tommy Jensen, Alek Vainshtein and the editors for a number of helpful comments. Research of both authors is supported in part by a Leverhulme Trust grant.

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Orlin and A.P. Punnen, A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.* 123: 75–102 (2002).

[2] N. Alon, G. Gutin and M. Krivelevich, Algorithms with large domination ratio. *J. Algorithms* 50: 118–131 (2004).

[3] N. Alon and J.H. Spencer, *The Probabilistic Method*, 2nd edition, Wiley, New York, (2000).

[4] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela and M. Protasi, *Complexity and Approximation*, Springer, Berlin, (1999).

[5] E. Bach and J. Shallit, *Algorithmic number theory, Volume 1*, MIT Press, Cambridge, Ma., (1996).

[6] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer-Verlag, London, (2000).

[7] J. Bang-Jensen, G. Gutin and A. Yeo, When the greedy algorithm fails. *Discerete Optimization*, to appear.

[8] D. Ben-Arieh, G. Gutin, M. Penn, A. Yeo and A. Zverovitch, Transformations of Generalized ATSP into ATSP: experimental and theoretical study. *Oper. Res. Lett.* 31: 357–365 (2003).

[9] C. Berge, *The Theory of Graphs*, Methuen, London, (1958).

[10] D. Berend and S.S. Skiena, Combinatorial dominance guarantees for heuristic algorithms. Manuscript, (2002).

[11] G. Cornuejols, M.L. Fisher and G.L. Nemhauser, Location of bank accounts to optimize float; an analytic study of exact and approximate algorithms. *Management Sci.* 23: 789–810 (1977).

[12] V.G. Deineko and G.J. Woeginger, A study of exponential neighbourhoods for the traveling salesman problem and the quadratic assignment problem, *Math. Prog. Ser. A* 87: 519–542 (2000).

[13] P. Erdös, On a lemma of Littlewood and Offord, *Bull. Amer. Math. Soc.* 51: 898–902 (1945).

[14] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multicriteria combinatorial optimization. *OR Spektrum* 22: 425–460 (2000).

[15] Ö. Ergun, J.B. Orlin and A. Steele-Feldman, Creating very large scale neighborhoods out of smaller ones by compounding moves: a study on the vehicle routing problem. Manuscript, (2002).

[16] F. Glover and A.P. Punnen, The traveling salesman problem: New solvable cases and linkages with the development of approximation algorithms, *J. Oper. Res. Soc.* 48: 502–510 (1997).

[17] G. Gutin. On an approach to solving the TSP. In *Proceedings of the USSR Conference on System Research,* pp. 184–185. Nauka, Moscow, (1984). (in Russian).

[18] G. Gutin, T. Jensen and A. Yeo, Domination analysis for minimum multiprocessor scheduling. Submitted.

[19] G. Gutin and A.P. Punnen, eds., *The Traveling Salesman Problem and its Variations*, Kluwer, Dordrecht, (2002).

[20] G. Gutin, A. Vainshtein and A. Yeo, Domination Analysis of Combinatorial Optimization Problems. *Discrete Appl. Math.* 129: 513–520 (2003).

[21] G. Gutin and A. Yeo, Polynomial approximation algorithms for the TSP and the QAP with a factorial domination number. *Discrete Appl. Math.* 119: 107–116 (2002).

[22] G. Gutin and A. Yeo, Upper bounds on ATSP neighborhood size. *Discrete Appl. Math.* 129: 533–538 (2003).

[23] G. Gutin and A. Yeo, Anti-matroids. *Oper. Res. Lett.* 30: 97–99 (2002).

[24] G. Gutin and A. Yeo, Introduction to domination analysis. Submitted.

[25] G. Gutin, A. Yeo and A. Zverovitch, Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discrete Appl. Math.* 117: 81–86 (2002).

[26] G. Gutin, A. Yeo and A. Zverovitch, Exponential Neighborhoods and Domination Analysis for the TSP. In *The Traveling Salesman Problem and its Variations* (G. Gutin and A.P. Punnen, eds.), Kluwer, Dordrecht, (2002).

[27] D.S. Johnson and L.A. McGeoch, The traveling salesman problem: A case study in local optimization. In *Local Search in Combinatorial Optimization* (E.H.L. Aarts and J.K. Lenstra, eds.), Wiley, Chichester, (1997).

[28] D.S. Johnson, G. Gutin, L. McGeoch, A. Yeo, X. Zhang, and A. Zverovitch, Experimental Analysis of Heuristics for ATSP. In *The Traveling Salesman Problem and its Variations* (G. Gutin and A. Punnen, eds.), Kluwer, Dordrecht, (2002).

[29] J. Håstad, Clique is hard to approximate within $n^{1-\epsilon}$, *Acta Mathematica* 182: 105–142 (1999).

[30] H. Kise, T. Ibaraki and H. Mine, Performance analysis of six approximation algorithms for the one-machine maximum lateness scheduling problem with ready times. *J. Oper. Res. Soc. Japan* 22: 205–223 (1979).

[31] A.E. Koller and S.D. Noble, Domination analysis of greedy heuristics for the frequency assignment problem. To appear in *Discrete Math.*

[32] C. Lund and M. Yannakakis, The approximation of maximum subgraph problems. *Proc. 20th Int. Colloq. on Automata, Languages and Programming*, Lect. Notes Comput. Sci. 700 (1993) Springer, Berlin, pp. 40–51.

[33] J. Oxley, *Matroid Theory*. Oxford Univ. Press, Oxford, (1992).

[34] A.P. Punnen, F. Margot and S.N. Kabadi, TSP heuristics: domination analysis and complexity. *Algorithmica* 35: 111–127 (2003).

[35] A. Punnen and S. Kabadi, Domination analysis of some heuristics for the asymmetric traveling salesman problem, *Discrete Appl. Math.* 119: 117–128 (2002).

[36] V.I. Rublineckii, Estimates of the accuracy of procedures in the Traveling Salesman Problem. *Numerical Mathematics and Computer Technology*, 4: 18–23 (1973) (in Russian).

[37] V.I. Sarvanov, On the minimization of a linear from on a set of all n-elements cycles. *Vestsi Akad. Navuk BSSR, Ser. Fiz.-Mat. Navuk* 4: 17–21 (1976) (in Russian).

[38] V.I. Sarvanov and N.N. Doroshko, Approximate solution of the traveling salesman problem by a local algorithm with scanning neighbourhoods of factorial cardinality in cubic time. In *Software: Algorithms and Programs.* Math. Institute of Belorussian Acad. of Sci., Minsk, 31: 11–13 (1981) (in Russian).

[39] T.W. Tillson, A hamiltonian decomposition of K^*_{2m}, $m \geq 8$. *J. Combinatorial Theory* B 29: 68–74 (1980).

[40] E. Zemel, Measuring the quality of approximate solutions to zero-one programming problems. *Math. Oper. Res.* 6: 319–332 (1981).

7

On Multi-Object Auctions and Matching Theory: Algorithmic Aspects

Michal Penn and Moshe Tennenholtz
Faculty of Industrial Engineering and Management
Technion, Israel Institute of Technology
Haifa 32000, Israel

Abstract

Auctions are the most widely used strategic game-theoretic mechanisms in the Internet. Auctions have been mostly studied from a game-theoretic and economic perspective, although recent work in AI and OR has been concerned with computational aspects of auctions as well. When faced from a computational perspective, combinatorial auctions are perhaps the most challenging type of auctions. Combinatorial auctions are auctions where buyers may submit bids for bundles of goods. Another interesting direction is that of constrained auctions where some restrictions are imposed upon the set of feasible solutions. Given that finding an optimal allocation of the goods in a combinatorial and/or constrained auction is in general intractable, researchers have been concerned with exposing tractable instances of combinatorial and constrained auctions problems. In this chapter we discuss the use of b-matching techniques in the context of combinatorial and constrained auctions.[1]

1. Introduction

The emergence of electronic commerce has led to increasing interest in the design of protocols for non-cooperative environments (see e.g. [24, 15, 29, 7]). The widespread proliferation of auctions in the Internet, and the fact that auctions are basic building blocks for a variety of economic protocols, have attracted many researchers to tackle the challenge of efficient auction design (e.g. [32, 19, 16, 28, 21]). The design of auctions introduces deep problems and challenges both from the game-theoretic and from the computational perspectives. This chapter mainly concentrates on computational aspects of auctions. More specifically, we concentrate on addressing computational problems of combinatorial and of constrained auctions, extending upon previous work on this

[1]Most of the results in this chapter are based on [22, 30, 10].

basic topic e.g., [26, 20, 28, 9, 22, 1]. Work on computational aspects of auctions fits into two main categories:

1. Finding tractable cases where one can find polynomial algorithm for the allocation of goods in complex auctions. For example, the work reported in [26] was the first to introduce such an approach in the context of combinatorial auctions, while in [22] this issue is tackled in the context of constrained multi-object auctions. In [20], the complexity of the bidding language is discussed as well.
2. Finding heuristics in order to practically tackle highly complicated auctions. Good examples on early work on that subject can be found in [28, 9].

In this chapter we deal with the first issue above. Our emphasis will be on the use of b-matching techniques in order to yield polynomial algorithms.

In a typical auction setting, an organizer designs a mechanism in order to sell k objects to a set of potential buyers. The outcome of each such mechanism is an allocation of the objects to the buyers, and a transfer of money from the buyers to the organizer. Much of the study of auctions is devoted to the case where there is only one object to be sold, see [33] for a survey, or (alternatively) to cases where the auctions for the different objects are independent of one another. However, many multi-object auctions do not satisfy the above-mentioned independence assumption. There are two major reasons for that: 1. The valuation of a buyer for a set of goods may be different from the sum of its valuations for each of them. 2. Auctions might be subject to various (social) constraints, which make only some of the allocations of the objects acceptable. For example, the famous FCC auction for radio spectrum [8], which inspired much of the recent work in auction theory, can be viewed as having both of the above properties.

Consider a situation where a VCR, a TV, and a Microwave are sold; a buyer may be willing to pay \$200 for the TV, \$300 for the VCR, and \$150 for the microwave, but might be willing to pay only \$550 for getting all of them; or in a game of Monopoly, a player may be willing to pay \$200 for a railroad, but would be willing to pay \$1000 for all four railroads. In order to allocate the goods in a satisfactory manner, bids for bundles of goods should be allowed; given these bids, we need to find an optimal, revenue maximizing, allocation of the goods. This problem is referred to as the *combinatorial auction problem*, and it is in general intractable [26]. As mentioned above, this chapter deals with polynomial algorithms for some tractable cases. A number of general techniques for tackling the complexity of combinatorial auctions have been introduced. Namely, several authors [20, 31] have dealt with the problem of winner determination in combinatorial auctions as an integer programming [IP] problem, and considered linear programming [LP] relaxations of that problem for isolating tractable cases of the general problem. Other researchers used dynamic programming techniques to solve some combinatorial auctions problem [13]. Another general technique for addressing the complexity of the combinatorial auction problems uses b-matching techniques [30, 10].

Considering the constrained auction problem, we assume that the buyers send independent bids for each of the objects, but the system needs to enforce some typical

constraints; the system may need to guarantee that a particular buyer (e.g., a representative of a certain minority) will get at least one good from a particular set of goods, or that a buyer will not get more than b goods from another set of goods (e.g., in order to guarantee that a single party will not take control of part of the market). We refer to the related auctions as *constrained multi-object auctions*. The computational aspects of the constrained auction problem have been dealt with very little in the literature [1, 22, 10]. Some special cases of this problem were also solved by using b-matching techniques [22, 10].

In this chapter, we survey the use of b-matching techniques for solving combinatorial and constrained auction problems. Thus, our goal may be seen as to equip researchers who deal with the theory and practice of combinatorial and constrained auctions with an additional general technique for addressing the complexity of that problem. Namely, we expose and explore the use of b-matching techniques for the combinatorial and constrained auctions set-up, and employ b-matching techniques in various ways in order to efficiently address several non-trivial instances of the combinatorial and constrained auctions problems. We will illustrate some of the b-matching techniques in some particular cases, and mostly for constrained auctions.

2. Preliminaries and Definitions

In combinatorial auction set-ups a seller sells m goods to n potential buyers. A bid of buyer i is a pair (S, p), where S is a bundle of goods and p is a non-negative real number that denotes the price offer for S. Let $X = \{x_1, x_2, \ldots, x_t\}$, where $x_i = (S_i, p_i)$, $1 \leq i \leq t$ be a set of bids, and denote by $S(x_i)$ and $P(x_i)$ the bundle of goods and the price offer of bid x_i, respectively. The *combinatorial auction problem* [CAP] is to find an $X_o \subseteq X$, for which $\Sigma_{X_o} P(x_i)$ is maximal, under the constraint that $S(x_i) \cap S(x_j) = \emptyset$ for every $x_i, x_j \in X_o, i \neq j$. The CAP is NP-hard, see [26], since it is equivalent to the weighted set packing problem which is known to be NP-complete [14]. The CAP is even hard to approximate [27].

The literature distinguishes between two types of combinatorial auctions. In a *sub-additive* combinatorial auction a buyer's bid for every bundle $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ of goods, is less than or equal to the sum of its bids for S_1 and S_2. In a *super-additive* combinatorial auction a buyer's bid for every bundle S of goods, as above, is greater than or equal to the sum of its bids for S_1 and S_2. To put it formally, let B be any set, then a function $f : 2^B \to \mathbb{R}$ is *sub-additive* if for any two disjoint subsets $S_1, S_2 \subseteq B$ it holds that $f(S_1 \cup S_2) \leq f(S_1) + f(S_2)$, *super-additive* if $f(S_1 \cup S_2) \geq f(S_1) + f(S_2)$ and *additive* if $f(S_1 \cup S_2) = f(S_1) + f(S_2)$. Typically, auctions for substitute goods are sub-additive, while auctions for complementary goods are super-additive. For example, if a buyer is interested in obtaining one TV from among a set of TVs, then typically his/her valuation for obtaining several TVs will be smaller than the sum of his/her valuations for obtaining each of them, since these goods are substitutes. Similarly, if a buyer is interested in purchasing a particular bundle of socks and shoes, then his/her valuation for the pair will be typically higher than the sum of his/her valuations for

each of the individual goods, since these goods are complementary. Hence, both auction types, the sub-additive and the super-additive ones, are of central importance.

As indicated before, the algorithms presented in this chapter are based on the general b-matching algorithm. Therefore, we give here the definition of the general b-matching problem. Let $G = (V, E)$ be a weighted undirected graph with $w : E \to \mathbb{R}$, its weight function. Let $b^{\mathrm{l}}, b^{\mathrm{u}} : V \to \mathbb{Z}_+$ (where $\mathbb{Z}_+$ is the set of non-negative integers) be two functions (the *lower* and *upper factor* functions), and let $\delta(v)$ denote the set of edges having vertex $v \in V$ as one of their end-vertices. A (*general*) *b-matching* is a function $y : E \to \mathbb{Z}_+$, such that $b^{\mathrm{l}}(v) \leq y(\delta(v)) \leq b^{\mathrm{u}}(v)$, where $y(\delta(v))$ stands for $\sum_{e\in\delta(v)} y(e)$, for each vertex v. Now, the *maximum* (*general*) *weighted b-matching problem* is: Given a weighted undirected graph (G, w) and two factor functions $b^{\mathrm{l}}, b^{\mathrm{u}} : V \to \mathbb{Z}_+$, find a (general) b-matching y with $\sum_{e\in E} w(e)y(e)$ as large as possible. Note that if $b^{\mathrm{l}} \equiv 0$, then the general b-matching problem is equivalent to the classic b-matching problem. In case no confusion arises, we refer to the general b-matching problem as the b-matching problem. A weighted b-matching of maximum value is called a *maximum b-matching*. We further define the capacitated general b-matching problem. Let $G = (V, E)$ be an undirected graph, and $c^{\mathrm{l}}, c^{\mathrm{u}} : E \to \mathbb{Z}_+$ (the *lower* and *upper capacity* functions), then a b-matching y is called *c-capacitated* if $c^{\mathrm{l}}(e) \leq y(e) \leq c^{\mathrm{u}}(e)$ for each edge e of G. The *capacitated weighted b-matching problem* is the following: Given a weighted undirected graph (G, w), two factor functions $b^{\mathrm{l}}, b^{\mathrm{u}} : V \to \mathbb{Z}_+$ and two capacity functions $c^{\mathrm{l}}, c^{\mathrm{u}} : E \to \mathbb{Z}_+$, find a c-capacitated b-matching y such that $\sum_{e\in E} w(e)y(e)$ is as large as possible. Note that this problem can be easily reduced to a similar problem with no lower capacity restrictions.

The above problem is known to be tractable, see for example [12], pp. 254–259. For the interested reader, we recommend further reading in [17], [11] pp. 135–224 and [4]; or any other book on combinatorial optimization. The interested reader can also consider [23] for a pseudopolynomial algorithm for the weighted b-matching problem, [6] for the first polynomial algorithm, and [2] for the first strongly polynomial algorithm. The tractability of the b-matching problem will play a significant role in this chapter.

In the simple case, where no constraints are imposed on the buyers, and the only limitations are the number of available goods, thus only simple quantity constraints are imposed on the goods, the problem can be solved by using a maximum capacitated b-matching algorithm in the following bipartite graph. The vertex bipartition is composed of the set of n buyers and the set of m goods. An edge connects a vertex representing a buyer to a vertex representing a good if the corresponding buyer submits a monetary bid on the corresponding good. The weight of this edge is the value of the bid. For illustration we give the following simple example.

Assume a combinatorial auction is taking place on a set of three types of goods that consists of two identical units of VCR, one unit of TV and one unit of Microwave. Assume, further, that the bids are given by 2 buyers and that each buyer is interested in at most one unit of each of the different goods. An agent's bid/valuation for any bundle of goods, one of each category, is taken as the sum of its bids for the individual items in this bundle. The first (second) buyer offers \$300 (\$250) for one

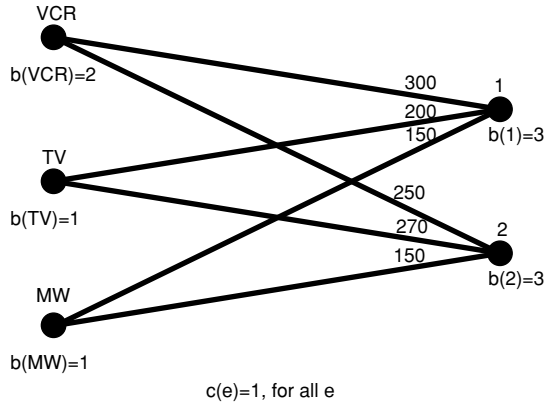


Figure 1. The constructed bipartite graph for the above example.

unit of VCR, \$200 (\$270) for the TV and \$150 (\$150) for the microwave. Then, the bipartite graph obtained is a complete bipartite graph where one side consists of three vertices that correspond to the three types of goods, labeled VCR,TV and MW, while the other side consists of two vertices, labeled 1 and 2, one for each buyer. Thus the bipartite graph obtained is a $K_{2,3}$. There are 6 edges in the graph with the following weights: $w(1, VCR) = 300$, $w(1, TV) = 200$, $w(1, MW) = 150$, $w(2, VCR) = 250$, $w(2, TV) = 270$ and $w(2, MW) = 150$, corresponding to the buyers' bids. For each vertex, the factor functions b^l, b^u are the same and are given by: $b(1) = b(2) = 3$, $b(VCR) = 2$ and $b(TV) = b(MW) = 1$. The lower and upper capacity functions are the same and equal to 1 for each edge since each buyer is willing to buy at most one unit of each good. Figure 1 below illustrates the above constructed bipartite graph. One can easily see that the set of edges $\{(1, VCR), (2, VCR), (2, TV), (1, MW)\}$ is a maximum matching of value 970. In our auction setting, the above solution will imply the following allocation of goods to buyers: buyer 1 will get a VCR and a Microwave while buyer 2 will get a VCR and a TV.

Let G be the above undirected bipartite graph that corresponds to the simple quantity constraints case and A its incidence matrix. It is well known that A is TUM (totally unimodular). This implies that the general maximum capacitated b-matching problem in G can be solved also by employing linear programming algorithm.

The importance of the above simple result is by showing a general connection between b-matchings and feasible allocations of the goods based on buyers' bids. In the sequel, we will consider also cases where this mapping is more subtle.

3. Combinatorial Auctions

In this section we deal with combinatorial auctions. The main feature of combinatorial auctions is that they allow handling cases where a buyer's valuation for a

bundle of goods may differ from the sum of its valuations for the individual items in that bundle.

3.1. Quantity Constraints

Consider an auction for the reservation of seats in a particular flight. Each potential buyer submits bids for each of the seats in the airplane, but restricts the total number of seats he may wish to obtain. This auction has the property that the payment of buyer i for the set of seats allocated to him, subject to his quantity constraint, is the sum of his bids for the individual seats in this set. However, this auction is a sub-additive combinatorial auction; a buyer will pay 0 for every additional seat assigned to him beyond his limit on the number of required seats.

Definition 3.1 *A* Quantity-constrained multi-object auction *is a sub-additive combinatorial auction where bids are of the form* $(a_1, p_1, a_2, p_2, \ldots, a_k, p_k, q)$ *where* p_i *is a price offer for object* a_i*, and* q *is the maximal number of objects that are to be assigned.*

Notice that in the above definition we used the term *multi-object auction*. This is in order to emphasize that although the auction is combinatorial, it has some syntactic similarity with other types of multi-object auctions, such as constrained multi-object auctions [22], since the bids are not stated explicitly for bundles of goods.

Theorem 3.1 *Quantity-constrained multi-object auctions are computationally tractable.*

3.2. Binary Bids

We have seen that simple quantity constraints can be incorporated into simple multi-object auctions, while still getting tractable solutions. Previous work has tried to tackle the tractability of combinatorial auctions where bids are given for non-singleton bundles. It was shown that the case of bundles of size two is tractable, while the case of larger bundles is NP-hard. Indeed, the first use of matching techniques in the context of combinatorial auctions appears in [26]. In that paper the authors consider the case of bundles of size two. They look at an undirected graph where the nodes represent the set of goods, and edges (and edge weights) are associated with bids. The solution of the CAP can be reduced to the computation of an optimal weighted matching. Matching techniques and the more powerful b-matching techniques can go much further. We now show that the case of bundles of size two and the case of quantity constraints can be tackled simultaneously in an efficient manner.

Definition 3.2 *A* quantity-constrained multi-object auction with binary combinatorial bundles *is a sub-additive combinatorial auction that allows two types of bids: 1. The bids allowed in a quantity-constrained multi-object auction. 2. Bids of the form* (a, p, b, q, l)

where p is the price offer for good a, q is the price offer for good b, and $p + q - l$ *is the price offer for the pair* $\{a, b\}$*, where* $0 < l < min(p, q)$*.*

Given the previous seats reservation example we allow each potential buyer to express sub-additive bids for pairs of seats, and the singletons they consist of, where she can explicitly declare the rebate asked for if both seats are allocated. This is in addition to bids that allow quantity constraints.

Using b-matching techniques it can be proven that:

Theorem 3.2 *Given a quantity-constrained multi-object auction with binary combinatorial bundles, the CAP is computationally tractable.*

3.2.1. Multi-Unit Binary Combinatorial Auctions

An interesting generalization of combinatorial auctions with binary bids is related to the case where there are several available units of each of the objects. In this case, when the buyer makes a bid for the bundle $\{a, b\}$, he does not care which copies of the objects a and b he obtains, as long as he obtains a copy of each one of them. In order to deal with this problem formally, we will need to introduce an extension of the CAP, termed multi-unit CAP (mu-CAP).

In a multi-unit combinatorial auction set-up a seller sells m goods $\{1, 2, \ldots, m\}$ to n potential buyers, where there are k_j units of good j $(1 \leq j \leq m)$.

A bid of buyer i is a pair (S, p), where S is a bundle of goods and p is a non-negative real number that denotes the price offer for S.[2] Let $X = \{x_1, x_2, \ldots, x_t\}$, where $x_i = (S_i, p_i)$ $(1 \leq i \leq t)$ be a set of bids, and denote by $S(x_i)$ and $P(x_i)$ the bundle of goods and the price offer of bid x_i, respectively. The *multi-unit combinatorial auction problem* [mu-CAP] is to find an $X_o \subseteq X$, for which $\Sigma_{X_o} P(x_i)$ is maximal, under the constraint that for every good j $(1 \leq j \leq m)$ $|X_o^j| \leq k_j$, where $X_o^j = \{x \in X_o : j \in S(x)\}$. Notice that in the mu-CAP we still allow each buyer to get at most one unit of each good, but the number of available units of good j is $k_j \geq 1$.

The mu-CAP fits many practical applications. For example, a retailer may wish to allow its customers to create their own desired binary bundle as part of a promotional sale, while limiting the number of units of each good available on that sale. As it turns out, b-matching allows to generalize the positive result on combinatorial auctions with binary bids [26] to the context of the mu-CAP in a quite straightforward manner.

Theorem 3.3 *The mu-CAP with binary bids is tractable.*

[2] This is a rather restricted version of multi-unit combinatorial auctions. In a more elaborated version a bid can ask for several units of each of the goods. For example, in such elaborated version we can give a bid for two TVs and three VCRs.

3.3. Beyond Binary Bids

As we mentioned, combinatorial auctions where bids are only for single goods or for pairs of goods are tractable [26]. However, when bids are for bundles of size greater than two, the CAP is in general intractable. Here we extend these results by considering cases of non-additive combinatorial auction, that is, where the bid for an allocated set of goods is different from the sum of bids for the singletons it consists of, and the bundles' sizes are greater than two. We start with two such results that we believe to be of considerable importance in this regard. The first result shows that bundles of size greater than two in which the bid for an allocated set of goods is different from the sum of bids for the singletons it consists of, can be handled in polynomial time. In the second result, a general form of combinatorial auctions where bids for triplets are permitted, i.e., combinatorial auctions with symmetric bids for triplets are shown to be tractable.

Definition 3.3 *An* almost-additive multi-object auction *is a combinatorial sub-additive auction where bids for non-singletons are of the form* $(a_1, p_1, a_2, p_2, \ldots, a_k, p_k, q)$ *where* p_i *is the price offer for object* a_i*, the price offer for any proper subset* $A \subset \{a_1, \ldots, a_k\}$ *equals* $\Sigma_{a_i \in A} p_i$*, and the offer for* $\{a_1, \ldots, a_k\}$ *is* q*; in addition,* $w = \Sigma_{1 \leq i \leq k} p_i - q > 0$*, and* $w < p_j$ $(1 \leq j \leq k)$*.*

In an almost-additive multi-object auction a shopping list of goods is gradually built until we reach a situation where the valuations become sub-additive; sub-additivity is a result of the requirement that $w > 0$. The other condition on w implies that the bid on the whole bundle is not too low with respect to the sum of bids on the single goods. Notice that typically q will be greater than the sum of any proper subset of the p_i's; our only requirement is that q will be lower than the sum of all the p_i's; hence, bidding on and allocation of the whole $\{a_1, a_2, \ldots, a_k\}$ bundle is a feasible and reasonable option. Such a situation may occur if each buyer is willing to pay the sum of the costs of the goods for any strict subset of the goods, but is expecting to get some reduction if he is willing to buy the whole set.

Theorem 3.4 *For an almost-additive multi-object auction, the CAP is computationally tractable.*

We continue with the case of combinatorial auctions with bids for triples of goods. The general CAP in this case is NP-hard. However, consider the following:

Definition 3.4 *A combinatorial auction with sub-additive symmetric bids for triplets is a sub-additive combinatorial auction where bids are either for singletons, for pairs of goods (and the singletons they are built of), or for triplets of goods (and the corresponding subsets). Bids for singletons and pairs of goods are as in Definition 3.2, while bids for triplets have the form* $(a_1, p_1, a_2, p_2, a_3, p_3, b_1, b_2)$*: the price offer for good* a_i *is* p_i*, the price offer for any pair of goods* $\{a_i, a_j\}$*,* $(1 \leq i, j \leq 3; i \neq j)$ *is* $p_i + p_j - b_1$*, and the price offer for the whole triplet* $\{a_1, a_2, a_3\}$ *is* $p_1 + p_2 + p_3 - b_2$*.*

Symmetric bids may be applicable to many domains. One motivation is the case where each buyer has a certain fixed cost associated with any purchase (e.g., paper work expenses, etc.), which is independent of the actual product purchased; this additional cost per product will decrease as a function of the number of products purchased (e.g., one does not need to duplicate the amount of paper work done when purchasing a pair of products rather than only one).

Theorem 3.5 *Given a combinatorial auction with sub-additive symmetric bids for triplets, where each bid for triplet* $(a_1, p_1, a_2, p_2, a_3, p_3, b_1, b_2)$ *has the property that* $b_2 > 3b_1$, *and* $p_i > b_2 - b_1 (1 \leq i \leq 3)$, *then the CAP is computationally tractable.*

The theorem makes use of the two conditions that connect b_1, b_2, and the bids on singletons. These conditions measure the amount of sub-additivity relative to the purely additive case where a bid for a bundle is the sum of bids for the singletons it consists of. The first condition is that the decrease in valuation/bid for a bundle, relative to the sum of bids for the singletons it consists of, will be proportional to the bundle's size; the second condition connects that decrease to the bids on the singletons, and requires that the above-mentioned decrease will be relatively low compared to the bids on the single goods. Both of these conditions seem quite plausible for many sub-additive auctions.

The technique for dealing with bundles of size 3 can be extended to bundles of larger size. However, the conditions on the amount of decrease in price offers as a function of the bundle size become more elaborated, which might make the result less applicable.

3.4. A Super-Additive Combinatorial Auction

In the previous sub-sections we have presented solutions for some non-trivial sub-additive combinatorial auctions. Here we show an instance of super-additive combinatorial auctions that can be solved by similar techniques.

Definition 3.5 *A combinatorial auction with super-additive symmetric bids for triplets is a super-additive combinatorial auction where each buyer submits a bid for a triplet of goods and its corresponding subsets, and is guaranteed to obtain at least one good. The bids for triplets have the form* $(a_1, p_1, a_2, p_2, a_3, p_3, b_1, b_2)$: p_i *is the price offer for good* a_i, *the price offer for any pair of goods* $\{a_i, a_j\} (1 \leq i, j \leq 3; i \neq j)$ *is* $p_i + p_j + b_1$, *and the price offer for the whole triplet* $\{a_1, a_2, a_3\}$ *is* $p_1 + p_2 + p_3 + b_2$.

Notice that the major additional restriction we have here is that the auction procedure must allocate at least a single good to each buyer. Of course, in practice the auction will have a reserve price, which will make this assumption less bothering for several applications. This is since it will require each buyer to pay at least the reserve price for the good it gets. For example, assume a car manufacturer wishes to promote a new set of car models by selling a set of cars in an auction to his buyers; it can decide

to guarantee that each participating buyer will be allocated at least one car but assign a reserve price; a car will not be sold for a price which is below that reserve price.

Theorem 3.6 *For combinatorial auctions with super-additive symmetric bids for triplets such that* $1.5b_1 \leq b_2 \leq 2b_1$*, the CAP is computationally tractable.*

3.5. Combinatorial Network Auctions

Linear goods refer to a set of goods where there is some linear order on them, i.e., they can be put on a line with a clear ordering among them. An example may be spots along a seashore. Linear goods turned out to be a very interesting setting for multi-object auctions. The assumption is that bids will refer only to whole intervals. For example, bids will refer only to a set of spots along the seashore that define an interval (with no "holes"). Auctions for linear goods are also a useful case of tractable combinatorial auctions (see [26, 20, 13]). Consider for example the case of discrete time scheduling of one resource (e.g., allocation of time slots in a conference room), or for the allocation of one-dimensional space (e.g., allocation of slots on a seashore), etc. When each request refers to a complete sequence of time slots, then by referring to each time slot as a good we get the case of auctions with linear goods. Another real-life example that fits also into this type of problem is that of radio frequency auctions, such as the FCC auction. For more on the FCC auction, see [18], [8] and [25]. In an auction for linear goods we have an ordered list of m goods, $g_1, \ldots, g_m$, and bids should refer to bundles of the form $g_i, g_{i+1}, g_{i+2}, \ldots, g_{j-1}, g_j$ where $j \geq i$, i.e., there are no "holes" in the bundle. The combinatorial auction problem, where bids are submitted on intervals is the same as auctions for linear goods, and is known as the *Interval Auction Problem*. This problem was first discussed by Rothkopf et al. in [26]. It was also studied by van Hoesel & Müller in [13]. A wide extension of the result on the tractability of auctions for linear goods is the following combinatorial network auctions problem.

Definition 3.6 *A network of goods is a network* $G(O) = (V(O), E(O))$*, where the set of nodes,* $V(O)$*, is isomorphic to the set of goods* $O = \{g_1, \ldots, g_m\}$*. A* combinatorial network auction *with respect to the set of goods* O *and the network* $G(O)$*, is a combinatorial auction where bids can be submitted only for bundles associated with paths in* $G(O)$*.*

Combinatorial network auction where $G(O)$ is a tree is termed a *combinatorial tree auction*. Combinatorial tree auction may be applicable in communication networks, where the underline graph is a tree. This is a reasonable assumption since in many cases the backbone of the network is a spanning tree. In such a case, goods are the edges of the tree and any message that should be delivered from vertex i to vertex j can be transmitted along the single path from i to j in the tree.

It is clear that combinatorial auctions for linear goods are simple instances of combinatorial tree auctions, where the tree is a simple path. Using, yet again, matching techniques, we can show:

Theorem 3.7 *Given a combinatorial tree auction problem, the CAP is computationally tractable.*

As trees are undirected graphs with no cycles, one might hope that the tractability of the CAP could be extended to acyclic directed graphs (DAG). However, this does not hold since the combinatorial network auction problem is NP-complete if $G(O)$ is a DAG. This NP-completeness result is obtained by using the following simple transformation to the general combinatorial auction problem. Consider a complete graph on n vertices. Assume the edges are oriented such that for any two vertices i and j, the edge between them is directed from i to j iff $i < j$. Then, it is easy to see that any bundle of goods can be represented by a path on the DAG, and vice versa, each path corresponds to the bundle of goods that consists of the goods represented by the vertices of the path.

4. Constrained Multi-Object Multi-Unit Auction

In constrained multi-object multi-unit auctions buyers' valuations for the different goods are taken as additive with the assumption that there are several identical units of each of the m different types of goods. For simplicity we omit the multi-object term of the constrained multi-object multi-unit auction problem and call it the constrained multi-unit auction problem. In the constrained multi-unit auction problem, there are external constraints on the allocation of goods. For example, consider a certain week in which a conference center has to organize a few conferences for several companies. Thus, the conference center aims to allocate his limited number of time slots of his conference rooms to the several companies in a somewhat balanced way. Hence, the number of time slots each company is allocated is restricted to obtain a feasible balanced allocation. This creates another, complementary type of non-trivial multi-object multi-unit auctions, that we now discuss. In general, the constrained multi-unit auction problem is NP-hard. This is proved by using a reduction of the 3SAT problem to a special version of the constrained multi-unit auction problem.

The constrained multi-unit auction problem is defined as follows. Denote by A the set of the seller and the n buyers, by T the set of the m types of goods and by q the quantity vector, with q_j units of good of type t_j, for all $j = 1, 2, \cdots, m$. In addition, denote by P the payment function with $p_{ij} \in \mathbb{R}$ being the bid submitted by buyer i to a **single** good of type t_j. Further on, some social constraints are imposed on the buyers. The following restrictions are considered. Buyer i is restricted to have at least γ_{ij}^{l} and at most γ_{ij}^{u} goods of type t_j, with γ_{ij}^{l}, γ_{ij}^{u} integers such that $0 \leq \gamma_{ij}^{\mathrm{l}} \leq \gamma_{ij}^{\mathrm{u}} \leq q_j$. In addition, the total number of goods to be allocated to buyer i is limited to be at least β_i^{l} and at most β_i^{u}, with β_i^{l}, β_i^{u} integers such that $0 \leq \beta_i^{\mathrm{l}} \leq \beta_i^{\mathrm{u}} \leq \sum_{j=1}^{m} q_j$. The constrained multi-unit auction problem is, therefore, the following:

Definition 4.1 *Given an 8-tuple* $(A, T, q, P, \beta^{\mathrm{l}}, \beta^{\mathrm{u}}, \gamma^{\mathrm{l}}, \gamma^{\mathrm{u}})$*, where A is the buyer set, T is the good type set,* $q \in \mathbb{Z}_+^m$ *is the good quantity vector,* $P : A \times T \to \mathbb{R}$ *is the payment function,*$\beta^{\mathrm{l}}, \beta^{\mathrm{u}} \in \mathbb{Z}_+^n$ *are the buyers' constraints vectors, and* $\gamma^{\mathrm{l}}, \gamma^{\mathrm{u}}$ *are the*

integer $n \times m$ buyer-good constraint matrices, find a feasible allocation of the goods to the buyers, which maximizes the seller's revenue.

Recall that the simple quantity constrained problem can be solved by constructing an appropriate simple bipartite graph and then running a general b-matching algorithm on that bipartite graph. In order to solve the constrained multi-unit auction problem, a somewhat more complicated transformation is needed for constructing an appropriate bipartite graph for this case that enables the use of a maximum capacitated general b-matching algorithm to solve the problem. Such a transformation exists and implies the following result.

Theorem 4.1 *The constrained multi-unit auction problem is computationally tractable.*

The constrained multi-unit auction problem can be generalized by further introducing entrance fees into the system. The generalized problem is termed the *constrained multi-unit auction problem with entrance fee*. That is, in addition to the constrained multi-unit auction problem definition, buyer i has to pay a fixed entrance fee of value $f_i \in \mathbb{R}$, if he wins at least one good. If entrance fees are to be paid just for taking part in the auction, then the optimal solution of the problem with entrance fees remains the same as that of the one without entrance fees, while its value increases by $\sum_{i=1}^{n} f_i$. However, if entrance fees are paid only by those buyers who wins at least one good, then employing a general b-matching algorithm to solve the problem requires further modification of the bipartite graph obtained for the constrained multi-unit auction problem. Such a modification exists and thus, the following theorem holds.

Theorem 4.2 *The constrained multi-unit auction problem with entrance fee is computationally tractable.*

The constrained multi-unit auction problem with entrance fee can be formulated naturally as a fixed charge integer programming (IP) problem. However, the LP relaxation of this IP formulation does not guarantee optimal integer solution. Inspired by the constructed bipartite graph for solving the constrained multi-unit auction problem with entrance fee, a more sophisticated IP formulation of the problem exists for which its LP relaxation guarantees optimal integer solutions. The integrality result follows from the fact that the incidence matrix of the obtained bipartite graph is TUM. This integrality result gives an alternative method for solving the problem.

4.1. Volume Discount

We have assumed up to this point that the monetary bids, p_{ij}, submitted by buyer i to each unit of good of type t_j, is independent of the size of the bid (order). Often, however, the seller is willing to consider charging less per unit for larger orders. The purpose of the discount is to encourage the buyers to buy the goods in large batches. Such volume discounts are common in inventory theory for many consumer goods. We believe that volume discounts are suitable, in a natural way, in many auction models as well.

There are many different types of discount policies. The two most popular are: all units and incremental. In each case, there are one or more breakpoints defining change points in the unit cost. There are two possibilities: either the discount is applied to all of the units in the bid (order), this is the *all-unit* discount, or it is applied only to the additional units beyond the breakpoint - the *incremental* discount. Here we consider the incremental case. We assume further that each buyer defines his own breakpoints.

Consider the following example. Two buyers are interested in trash bags and their bids are of the following price policy. The first buyer is willing to pay 30 cents for each bag for quantities below 500; for quantities between 500 and 1,000, he is willing to pay 30 cents for each of the first 500 bags and for each bag of the remaining amount he is willing to pay 29 cents each; for quantities over 1,000, for the first 500 bags he is willing to pay 30 cents each, for the next 500 bags 29 cents each, and for the remaining amount he is willing to pay 28 cents each. The second buyer has the following similar policy. He is willing to pay 31 cents for each bag for quantities below 600; for quantities over 600 bags he is willing to pay 31 cents for each of the first 600 bags and for the remaining amount he is willing to pay 28 cents for each bag. The breakpoints in this example are: 500 and 1,000 for the first buyer and 600 for the second one.

Formally the incremental volume discount constrained multi-unit auction problem is defined as follows. Let r_{ij} denote the number of breakpoints defined by buyer i for a good of type t_j, and let d_{ij}^s denote the values of the breakpoints, with $d_{ij}^0 = 0$. Also, let p_{ij}^s denote the unit cost for any additional unit beyond the breakpoint d_{ij}^{s-1}, for $1 \leq s \leq r_{ij}$. Then, each bid is composed of a 2-dimensional vector of length r_{ij}, (p_{ij}^s, d_{ij}^s). That is, buyer i will pay p_{ij}^1 for each one of the first d_{ij}^1 goods of type t_j allocated to him. He will further pay p_{ij}^2 for each one of the next $d_{ij}^2 - d_{ij}^1$ goods, p_{ij}^3 for each one of the next $d_{ij}^3 - d_{ij}^2$ goods, etc. The above mentioned problem is denoted as the *incremental volume discount constrained multi-unit auction problem with entrance fee* . The following assumptions are assumed: $p_{ij}^1 > p_{ij}^2 > \cdots > p_{ij}^{r_{ij}}$ and $0 = d_{ij}^0 < d_{ij}^1 < d_{ij}^2 < \cdots < d_{ij}^{r_{ij}}$.

Using, yet, even more complicated construction, an appropriate bipartite graph can be obtained so that running a general capacitated b-matching algorithm on this graph will result a solution to the incremental volume discount constrained multi-unit auction problem with entrance fee. Hence, the following theorem follows.

Theorem 4.3 *The incremental volume discount constrained multi-unit auction problem with entrance fee is computationally tractable.*

5. Conclusion

In this chapter we discussed the use of b-matching techniques in order to handle non-trivial multi-object multi-unit auctions. We have shown that b-matching techniques are highly applicable for both combinatorial auctions and constrained auctions, two fundamental and complementary classes of multi-object and multi-unit auctions. While

the more standard approach for handling complex auctions, such as combinatorial auctions, typically relies on straightforward techniques such as LP relaxation of the corresponding IP problem, *b*-matching techniques can be used in order to address additional types of multi-object auctions. By combining standard matching techniques, more general *b*-matching techniques, and capacitated *b*-matching techniques, tractable solutions are found to basic combinatorial and constrained auctions.

The research reported in this chapter can be extended in various directions. First, combinatorial bids and constraints are two basic properties of multi-object auctions, and in future work one can combine them in a shared framework. Given that framework, a study of the applicability of *b*-matching techniques to the more general setting should be introduced. In addition, we believe that the applicability of *b*-matching techniques goes beyond auctions, and may be useful in other economic settings. For example, it seems that these techniques can be used in the context of double auctions and exchanges. Finally, we remark that matching is a basic ingredient of markets in general; therefore, a more general theory connecting algorithmic matching theory and economic interactions may be a subject of future research.

References

[1] K. Akcoglu, J. Aspnes, B. DasGupta and M. Kao, Opportunity cost algorithms for combinatorial auctions. Technical Report 2000-27, DIMACS, (2000).

[2] R. P. Anstee, A polynomial algorithm for *b*-matching: An alternative approach, *Information Processing Letters* 24: 153–157 (1987).

[3] S. A. Cook, The complexity of theorem-proving procedures, *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, Association for Computing Machinery, New-York, (1971) pp. 151–158.

[4] W. Cook, W. R. Pulleyblank, Linear systems for constrained matching problems, *Mathematics of Operations Research* 12: 97–120 (1987).

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to algorithms, The MIT Press, Twentieth printing, (1998).

[6] W. H. Cunningham and A. B. March III, A primal algorithm for optimum matching, *Mathematical Programming Study* 8: 50–72 (1978).

[7] E. Durfee, What your computer really needs to know, you learned in kindergarten. In *10th National Conference on Artificial Intelligence,* (1992) pp. 858–864.

[8] Federal communications commission (FCC) at http://www.fcc.gov/auctions.

[9] Y. Fujishima, K. Leyton-Brown, Y. Shoham, Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches, *International Joint Conference on Artificial Intelligence (IJCAI),* Stockholm, Sweden, (1999).

[10] R. Frank and M. Penn, Optimal allocations in constrained multi-object and combinatorial auctions, in preparation.

[11] A. M. H. Gerards, in *Handbooks in Operations Research and Management Science* Vol. 7, ed. M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser, "Matching" in Network Models, (1995) pp. 135–212.

[12] M. Grotschel, L. Lovasz, A. Schrijver, Geometric algorithms and combinatorial optimization, Springer-Verlag, Second Corrected Edition, (1993).

[13] S. van Hoesel, R. Müller, Optimization in electronic markets: Examples in combinatorial auctions, *Netonomics*, 3: 23–33 (2001).

[14] R. Karp, Reducibility among combinatorial problems, in R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, Plenum Press, NY, (1972) pp. 85–103.

[15] S. Kraus, Negotiation and cooperation in multi-agent environments, *Artificial Intelligence*, 94: 79–97 (1997).

[16] D. Lehmann, L.I. O'Callagham and Y. Shoham, Truth revelation in rapid, approximately efficient combinatorial auctions, In *ACM Conference on Electronic Commerce*, (1999) pp. 96–102.

[17] L. Lovász and M. D. Plummer, *Matching Theory*, Akadémiai Kiadó, Budapest (North Holland Mathematics Studies Vol. 121, North Holland, Amsterdam, 1986).

[18] J. McMillan, Selling spectrum rights, *Journal of Economic Perspective* 8: 145–162 (1994).

[19] D. Monderer and M. Tennenholtz, Optimal auctions revisited *Artificial Intelligence*, 120: 29–42 (2000).

[20] N. Nisan, Bidding and allocation in combinatorial auctions, in *ACM Conference on Electronic Commerce*, (2000) pp. 1–12.

[21] D. C. Parkes, ibundel: An Efficient Ascending Price Bundle Auction, in *ACM Conference on Electronic Commerce*, (1999) pp. 148–157.

[22] M. Penn, M. Tennenholtz, Constrained Multi-Object Auctions, *Information Processing Letters* 75: 29–34 (2000).

[23] W. R. Pulleyblank, Facets of matching polyhedra, Ph.D. Thesis, University of Waterloo, Department of Combinatorial Optimization, Canada, (1973).

[24] J. S. Rosenschein and G. Zlotkin, *Rules of Encounter*, MIT Press, (1994).

[25] A. Roth, FCC Auction (and related matters): Brief Bibliography, in A. Roth's homepage, Harvard Economics Department and Harvard Business School, USA.

[26] M. H. Rothkopf, A. Pekeč, R. M. Harstad, Computationally Manageable Combinational Auctions, *Management Science* 44: 1131–1147 (1998).

[27] T. Sandholm, Limitations of the Vickery auction in computational multiagent systems, *Proceedings of the Second International Conference on Multiagent Systems*, Keihanna Plaza, Kyoto, Japan, (1996) pp. 299–306.

[28] T. Sandholm, An Algorithm for Optimal Winner Determination in Combinatorial Auctions, *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 542–547, Stockholm, Sweden, (1999).

[29] M. Tennenholtz, Electronic commerce: From game-theoretic and economic models to working protocols. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1420–1425 Stockholm, Sweden, (1999).

[30] M. Tennenholtz, Some Tractable Combinatorial Auctions *Artificial Intelligence*, 140: 231–243 (2002).

[31] S. de Vries, R. V. Vohra, Combinatorial Auctions: A Brief Survey, unpublished manuscript.

[32] M. P. Wellman, W. E. Walsh, P. R. Wurman and J. K. MacKie-Mason, Auction Protocols for Decentralized Scheduling, *Games and Economic Behavior*, 35: 271–303 (2001).

[33] E. Wolfstetter, Auctions: An Introduction, *Journal of Economic Surveys*, 10: 367–420 (1996).

8

Strategies for Searching Graphs

Shmuel Gal
The University of Haifa

1. Introduction

The search problems considered in this chapter involve detecting an object such as a person, a vehicle, or a bomb hiding somewhere in a graph (on an edge or at a vertex). Some natural examples for such problems can be quite intriguing like finding a strategy to get to the exit point of a maze in minimum expected time. We shall see that the optimal search strategies sometimes have a simple structure, but, on the other hand, some very simple graphs lead to difficult problems with complicated solutions.

An extensive research effort, initiated by Koopman and his colleagues in 1946, has been carried out on this type of problem. (An updated edition of his book "Search and Screening" has appeared in 1980.) Most earlier works on search assume that there exists a probability distribution for the location of the target, which is known to the searcher (i.e., a Bayesian approach is used). In addition, most previous works, such as the classic work of Stone [29] "Theory of Optimal Search" which was awarded the 1975 Lanchester Prize by the Operations Research Society of America, are concerned with finding the optimal distribution of effort spent in the search but do not actually present optimal search trajectories. A comprehensive survey was presented by Benkoski, Monticino, and Weisinger [3].

In this chapter we shall take into consideration the fact that usually the searcher, or searchers, have to move along a continuous trajectory and attempt to find those trajectories which are optimal in the sense described in the sequel. We shall usually not assume any knowledge about the probability distribution of the target's location, using instead a minimax approach. The minimax approach can be interpreted in two ways. One could either decide that because of the lack of knowledge about the distribution of the target, the searcher would like to assure himself against the worst possible case; or, as in many military situations, the target is an opponent who wishes to evade the searcher as long as possible. This leads us to find worst case solutions for the following scenario. The search takes place in a set Q to be called the *search space* and the searcher usually starts moving from a specified point O called the origin and is free to

choose any continuous trajectory inside Q, subject to a maximal velocity constraint. The searcher continues to move until he finds the (immobile) target. The aim of the searcher is to minimize the (expected) time spent in searching. The search spaces considered are graphs. Many search problems in two (or more) dimensional search spaces are analyzed in [12] and [1].

We thus use the following framework. A searcher moves with unit speed along a *continuous trajectoryS* looking for a (hidden) target in a graph Q. The notion "graph" means, here, any finite connected set of arcs which intersect only at nodes of Q. Thus, Q can be represented by a set in a three-dimensional[1] Euclidean space with vertices consisting of all points of Q with degree $\neq 2$ plus, possibly, a finite number of points with degree 2. (As usual, the *degree* of each node is defined as the number of arcs incident at this node.) Note that we allow more than one arc to connect the same pair of nodes.

The sum of the lengths of the arcs in Q will be denoted by μ and called the *total length* or the *measure* of the graph.

We consider searching for a target which can be located at any point, H, of the graph, where H is either a vertex or any point on one of the arcs of Q. A hiding distribution h is a probability measure on Q. A pure search strategy S is a continuous trajectory in Q, starting at the origin O, with speed 1. For any specific time t, $S(t)$ denotes the location of the searcher at time t. A mixed search strategy s is a probability measure on the set of these trajectories. (For a more formal presentation see [12] or [1].)

We assume that the searcher finds the target as soon as his search trajectory visits the target location. Obviously, the searcher would like to minimize the time spent in searching (the *search time) T*. We will use a worst case analysis concentrating on optimal search strategies that assure finding the target in minimum expected time. Thus

Definition 1 *The value $v(s)$ of a search strategy s is the maximum expected search time taken over all the hiding points.*

Definition 2 *For any search problem*

$$v = \inf_s v(s)$$

is called the "search value" or simply the "value" (of the search problem).
A strategy $\bar{s}$ which satisfies

$$v(\bar{s}) = v$$

is called "an optimal search strategy".

[1] Two dimensions are sufficient for planar graphs.

Remark 3 *The value v is actually the value of the search game between a searcher and an immobile hider (the target). It is proven in [12] that this game always has a value and an optimal search strategy.*

We will not consider problems in which the target can move and try to evade the searcher. Such problems, as well as many other search games, are analyzed in [1].

Remark 4 *Except for trivial cases, if only pure (nonrandomized) search strategies are used, then several searchers are needed to guarantee capturing a mobile evader. The interesting problem of determining the minimal number of such searchers in a given network, called the 'search number', was considered by Parsons [24,25] and Megiddo and Hakimi [22]. This problem has attracted much research. An important result was obtained by Megiddo et al. [23] who showed that the problem of computing the search number is NP-hard for general graphs but can be solved in linear time for trees.*

In the next section we derive general upper and lower bounds for the expected search time. We shall have cases, such as Eulerian graphs (Theorem 16 and 17) in which the searcher can perform the search with "maximum efficiency" which assures him a value of $\mu/2$. On the other hand, in the case of a non-Eulerian graph, we shall prove that the value is greater than $\mu/2$ and that the maximum value is μ. This value is obtained in the case that Q is a tree (Section 3). A more general family which contains both the Eulerian graphs and the trees as subfamilies is the family of *weakly Eulerian* graphs (see Definition 27) for which the optimal search strategy has a simple structure similar to that for Eulerian graphs and trees (Section 4). We shall also demonstrate the complications encountered in finding optimal strategies in the case that Q is not weakly Eulerian, even if the graph simply consists of three unit arcs connecting two points (Section 5). Finally we describe in Section 6 optimal strategies for searching a graph with no a-priori knowledge about its structure. This is often called *traversing a maze*.

Except for short proofs we present only very general sketches for our results. The full proofs, as well as many other search games can be found in [1].

2. Lower and Upper Bounds

A useful hiding distribution for obtaining lower bounds on the expected search time is the uniform distribution h_μ, i.e., "completely random" hiding strategy on Q. More precisely:

Definition 5 *The uniform distribution h_μ is a random choice of the target location H such that for all (measurable) sets $B \in Q$,*

$$\Pr(H \in B \mid h_\mu) = \mu(B)/\mu.$$

(Note that $\mu \equiv \mu(Q)$.)

The next theorem presents a useful lower bound for v.

Theorem 6 *If the target location is chosen according to the uniform distribution h_μ then, for any search strategy, the expected time spent in searching is at least $\mu/2$.*

Proof. It is easy to see that the probability that the target has been found by time t satisfies

$$\Pr(T \le t) \le \min\left[\frac{t}{\mu}, 1\right]$$

which implies that

$$\begin{aligned} E(T) &= \int_0^\infty \Pr(T > t)\, dt \ge \int_0^\infty \max\left[1 - \frac{t}{\mu}, 0\right] dt \\ &= \int_0^\mu \left(1 - \frac{t}{\mu}\right) dt = \mu/2. \end{aligned}$$

■

Since the hiding distribution h_μ guarantees $\mu/2$ against any starting point of the searcher, we also have the following:

Corollary 7 *If there exists a search strategy with value $v = \mu/2$, then allowing the searcher to choose his starting point does not reduce the expected search time below $\mu/2$.*

In studying search trajectories in Q, we shall often use the notion "closed trajectory" defined as follows.

Definition 8 *A trajectory $S(t)$ defined for $0 \le t \le \tau$ is called "closed" if $S(0) = S(\tau)$. (Note that a closed trajectory may cut itself and may even go through some of the arcs more than once.) If a closed trajectory visits all the points of Q, then it is called a "tour".*

Eulerian tours, defined as follows, will have an important role in our discussion.

Definition 9 *A graph Q is called "Eulerian" if there exists a tour L with length μ, in which case the tour L will be called an "Eulerian tour".*
A trajectory $S(t)$, $0 \le t \le \mu$, which covers all the points of Q in time μ, will be called an "Eulerian path". (Such a path may not be closed.)

It is well known that Q is Eulerian if and only if the degree of every node is even, and that it has an Eulerian path if and only if at most two nodes have odd degree. In

the case that Q has two odd degree nodes, then every Eulerian path has to start at one of them and end at the other. (See [9].)

Chinese postman tours will play an important role in our discussion:

Definition 10 *A closed trajectory which visits all the points of Q and has minimum length will be called a "minimal tour" (or a "Chinese postman tour") and is usually denoted by L. Its length will be denoted by $\bar{\mu}$.*

Finding a minimal tour for a given graph is called the *Chinese postman* problem. This problem can be reformulated for any given graph Q as follows. Find a set of arcs, of minimum total length, such that when these arcs are duplicated (traversed twice in the tour) the degree of each node becomes even. This problem was solved by Edmonds [6] and Edmonds and Johnson [7] using a matching algorithm which uses $O(n^3)$ computational steps, where n is the number of nodes in Q. This algorithm can be described as follows. First compute the shortest paths between all pairs of odd-degree nodes of Q. Then partition the odd-degree nodes into pairs so that the sum of lengths of the shortest paths joining the pairs is minimum. This can be done by solving a minimum weighted matching problem. The arcs of Q in the paths identified with arcs of the matching are the arcs which should be duplicated (i.e., traversed twice). The algorithm is also described by Christofides [5] and Lawler [21]. (An updated survey on the Chinese postman problem is presented by Eiselt et al. [8].)

Once an Eulerian graph is given, one can use the following simple algorithm for finding an Eulerian tour (see [4]). Begin at any node A and take any arc not yet used as long as removing this arc from the set of unused arcs does not disconnect the graph consisting of the unused arcs and incident nodes to them. Some algorithms which are more efficient than this simple algorithm were presented by Edmonds and Johnson [7]. Actually, it is possible to slightly modify Edmond's algorithm in order to obtain a trajectory (not necessarily closed) which visits all the points of Q and has minimum length. This trajectory is a minimax search trajectory (pure search strategy).

Example 11 *Consider the graph in Figure* 1 *(having the same structure as Euler's Köninsberg bridge problem). The duplicated arcs in the minimal tour can be either* $\{a_1, b_1, c_1\}$ *(based on the partition* $\{AB, OC\}$ *) or* $\{b_1, d\}$ *(based on* $\{OB, AC\}$ *) or* $\{a_1, e\}$ *(based on* $\{OA, BC\}$ *). The corresponding sum of the lengths of the arcs is either* 5 *or* 4 *or* 5. *Thus, the minimal tour duplicates the arcs b_1 and d. The minimal tour can be traversed by the following trajectory S_1*

$$S_1 = Ob_1Bb_1Ob_2BeCdAa_1Oa_2AdCc_1O \tag{1}$$

with length $\bar{\mu} = 20$.

A minimax search trajectory is based on duplicating arcs (having minimum total length) in order to make all the node degrees even except for the starting point O plus

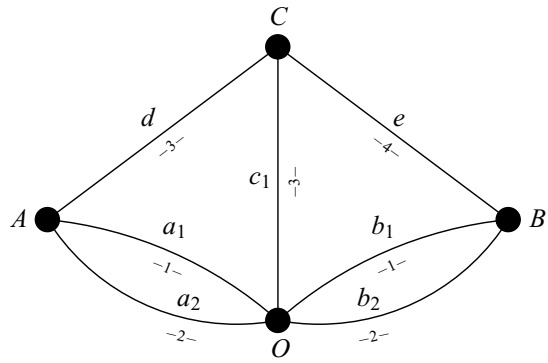


Figure 1.

another node. It can be easily seen that the arcs that have to be duplicated are a_1 *and* b_1 *leading to the following minimax trajectory*

$$Ob_1Bb_1Ob_2BeCdAa_1Oa_1Aa_2Oc_1C$$

with length 18.

The length of a Chinese postman tour is bounded by twice the size of the search space as is stated by the following:

Lemma 12 *Any minimal tour satisfies* $\bar{\mu} \leq 2\mu$. *Equality holds only for trees.*

This result can easily be obtained by considering a graph $\bar{Q}$ obtained from Q as follows. To any arc b in Q, add another arc $\bar{b}$ which connects the same nodes and has the same length as b. It is easy to see that $\bar{Q}$ is Eulerian with a tour length 2μ. If we now map the graph $\bar{Q}$ into the original graph Q such that both arcs b and $\bar{b}$ of $\bar{Q}$ are mapped into the single arc b of Q, then the tour $\bar{L}$ is mapped into a tour L of Q with the same length 2μ. If Q is not a tree, it contains a circuit C. If we remove all new arcs $\bar{b}$ in $\bar{Q}$ corresponding to this circuit, then the resulting graph is still Eulerian and contains Q, but has total length less than 2μ.

Using the length $\bar{\mu}$ of the minimal tour, we now derive an upper bound for the value of the search game in a graph in terms of the length of its minimal tour.

Lemma 13 *For any graph* Q, *the value* v *satisfies* $v \leq \bar{\mu}/2$, *where* $\bar{\mu}$ *is the length of a minimal tour* L *of* Q.

Proof. Assume that the searcher uses a strategy $\bar{s}$ which encircles L equiprobably in each direction. It can be easily seen that, for any target location, the sum of these two possible discovery times is at most $\bar{\mu}$. Consequently, the expected discovery time does not exceed $\bar{\mu}/2$. ■

Definition 14 *The search strategy $\bar{s}$ used for proving Lemma 13 will be called the "random Chinese postman tour".*

A random Chinese tour of Figure 1 is either to randomly follow either S_1 (see (1)) or the same path in the opposite direction:

$$Oc_1CdAa_2Oa_1AdCeBb_2Ob_1Bb_1O.$$

Combining Theorem 6, Lemma 12 and Lemma 13 we obtain the following result.

Theorem 15 *For any graph Q, the search value v satisfies*

$$\mu/2 \leq v \leq \bar{\mu}/2 \leq \mu. \tag{2}$$

In the next section we will show that the upper bound, μ, is attained if and only if Q is a tree.

The lower bound, $\mu/2$, is attained if and only if Q is Eulerian because if Q is Eulerian then $\bar{\mu} = \mu$ so that (2) implies:

Theorem 16 *If Q is Eulerian then the Random Chinese Postman tour is an optimal search strategy yielding a value of $\mu/2$ (half the total length of Q).*

Actually the Eulerian graphs are the only graphs with a search value of $\mu/2$:

Theorem 17 *If the search value of a graph Q is $\mu/2$ then Q is Eulerian*

The proof is given in [12] and in [1].

Corollary 18 *For an Eulerian graph, the optimal strategies and the value of the search problem remain the same if we remove the usual restriction $S(0) = O$ and instead allow the searcher an arbitrary starting point.*

The claim of the corollary is an immediate consequence of Corollary 7.

Remark 19 *Corollary 18 fails to hold for non-Eulerian graphs because (unlike the Eulerian case) the least favorable hiding distribution usually depends on the starting point of the searcher.*

In general, if we allow the searcher an arbitrary starting point, then $v = \mu/2$ if and only if there exists an Eulerian path (not necessarily closed) in Q. (If there exists such a path then the searcher can keep the expected capture time $\leq \mu/2$ by an analogous strategy to $\bar{s}$ of Lemma 13. If there exists no Eulerian path in Q, then the uniform hiding distribution yields the expected search time exceeding $\mu/2$.)

3. Search on a Tree

We now consider searching on a tree. In this case we will see that the search value is simply the total length of the tree ($v = \mu$), and that a random Chinese postman tour is an optimal search strategy.

The fact that $v \leq \mu$ is an immediate consequence of Lemma 12 and Lemma 13. The reverse inequality is more difficult to establish. First observe that if x is any point of the tree other than a terminal node, the subtree Q_x (the union of x and the connected component, or components, of $Q \setminus \{x\}$ which doesn't contain the starting point O) contains a terminal node y. Since no trajectory can reach y before x, hiding at y strictly dominates hiding at x. Since we are interested in worst cases, we may restrict the target's distributions to those concentrated on terminal nodes.

To give a hint on the worst case hiding distribution over the terminal nodes, we first consider a very simple example. Suppose that Q is the union of two trees Q_1 and Q_2 who meet only at the starting node O. Let μ_i denote the total length of Q_i. Let p_i denote the probability that the hider is in the subtree Q_i and assume that the searcher adopts the strategy of first using a random Chinese postman strategy in Q_1, and then at time $2\mu_1$ starts again from O to use a random Chinese postman strategy on Q_2. The expected search time resulting from such a pair of strategies can be obtained by Algorithm 20, which will be described later, giving

$$p_1\mu_1 + p_2(2\mu_1 + \mu_2) = \mu_1 + p_2(\mu_1 + \mu_2).$$

Conducting the search in the opposite order gives an expected capture time of

$$\mu_2 + p_1(\mu_1 + \mu_2).$$

Consequently, if the p_i are known, the searcher can ensure an expected capture time of

$$\min\left[\mu_1 + p_2(\mu_1 + \mu_2), \mu_2 + p_1(\mu_1 + \mu_2)\right].$$

Since the two expressions in the bracket sum to $2(\mu_1 + \mu_2)$, it follows that the hider can ensure an expected capture time of at least $\mu_1 + \mu_2$ only if these expressions are equal, or

$$p_1 = \frac{\mu_1}{\mu_1 + \mu_2},\ p_2 = \frac{\mu_2}{\mu_1 + \mu_2}\ .$$

This analysis shows that if $v = \mu$ then a worst case target distribution must assign to each subtree a probability proportional to its total length.

In general, a least favorable target distribution will be constructed recursively as follows.

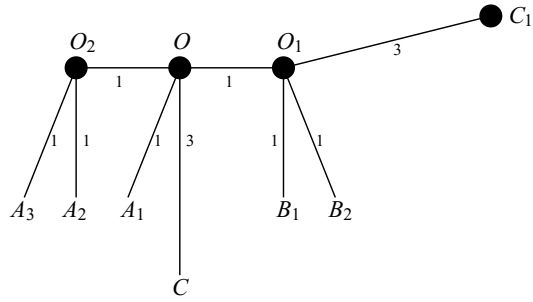


Figure 2.

Algorithm 20 *First recall our above argument that the hiding probabilities are positive only for the terminal nodes of Q . We start from the origin O with $p(Q) = 1$ and go towards the leaves. In any branching we split the probability of the current subtree proportionally to the measures of subtrees corresponding to the branches. When only one arc remains in the current subtree we assign the remaining probability, $p(A)$, to the terminal node A at the end of this arc.*

We illustrate this method for the tree depicted in Figure 2.

From O we branch into A_1, C, O_1 and O_2 with proportions 1, 3, 6 and 3, respectively. Thus, the probabilities of the corresponding subtrees are $\frac{1}{13}$, $\frac{3}{13}$, $\frac{6}{13}$ and $\frac{3}{13}$, respectively. Since A_1 and C are leaves we obtain $p(A_1) = \frac{1}{13}$ and $p(C) = \frac{3}{13}$. Continuing towards O_1 we split the probability of the corresponding subtree, $\frac{6}{13}$, with proportions $\frac{1}{5}$, $\frac{1}{5}$ and $\frac{3}{5}$ between B_1, B_2 and C_1 so that:

$p(B_1) = \frac{6}{65}$, $p(B_2) = \frac{6}{65}$ and $p(C_1) = \frac{18}{65}$.

Similarly $p(A_2) = \frac{3}{13} \times \frac{1}{2} = \frac{3}{26}$ and $p(A_3) = \frac{3}{13} \times \frac{1}{2} = \frac{3}{26}$.

In order to show that $v = \mu$ it has to be proven that the probability distribution generated by Algorithm 20 is least favorable, *i.e.*, guarantees an expected search time of at least μ. This proof begins with the following result.

Lemma 21 *Consider the two trees Q and Q' as depicted in Figure 3. The only difference between Q and Q' is that two adjacent terminal branches BA_1 of length a_1 and BA_2 of length a_2 (in Q) are replaced by a single terminal branch BA' of length $a_1 + a_2$ (in Q'). Let v be the value of the search game in Q and let v' be the value of the search game in Q'. Then $v \geq v'$.*

Proof. We present a sketch of the proof of the lemma based on the following construction: Let h' be a least favorable target distribution for the tree Q', so that h' guarantees an expected search time of at least v', for any pure trajectory S'.

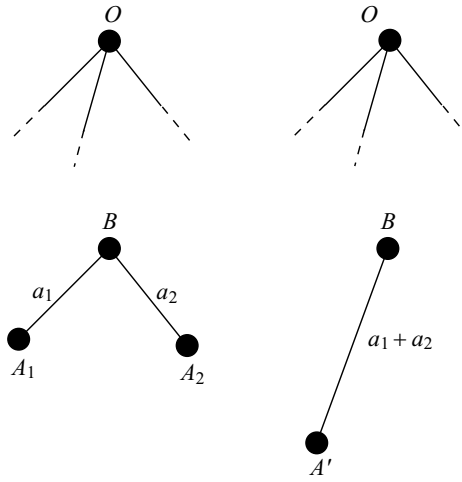


Figure 3.

We may assume, as explained above, that h' is concentrated on terminal nodes. Given h', we construct a target distribution h in the graph Q as follows: For any node other than A_1 or A_2, the probability is the same for h and h'. The probabilities $p_1 = h(A_1)$ and $p_2 = h(A_2)$ of choosing A_1 and A_2 when using h are given by the formulae

$$p_1 = \frac{a_1}{a_1 + a_2} p' \quad \text{and} \quad p_2 = \frac{a_2}{a_1 + a_2} p' \tag{3}$$

where $p' = h'(A')$ is the probability of A' being chosen by h', and a_1, a_2 are the lengths defined in the statement of the Lemma (see Figure 3). It can be shown $v \geq v'$ by proving that for any search trajectory S in Q the expected search time against the distribution h is at least v'. ■

Using Lemma 21, we can readily establish by induction on the number of terminal nodes that the target distribution of Algorithm 20 indeed guarantees an expected search time $\geq \mu$ for any tree. Thus we have:

Theorem 22 *Q is a tree if and only if $v = \mu$.*

The optimal search strategy is the random Chinese postman tour.
A worst case hiding distribution can be constructed recursively using Algorithm 20.

Remark 23 *An interesting model is one in which the target is hidden at a node of a graph and the search cost is composed of the travel cost plus c_i - the cost of inspecting the chosen node i. The case of linear ordered nodes is solved in [16] and [17]. The more general case of a tree is solved in [19] and [18].*

It should be noted that if $c_i = c$ for all i then the problem can be transformed into our standard search on a tree (with no inspection costs) by constructing the following tree: To each node add an arc of length $c/2$, with a new leaf at the other end of this arc. The optimal strategies for the equivalent tree are optimal for the problem with c inspection cost but the expected (optimal) cost is reduced by $c/2$. (If c_i are not all equal then adding similar arcs, with length $c_i/2$ to each node, leads to a close but not identical problem.)

4. When is the Random Chinese Postman Tour Optimal?

In the case that the graph Q is neither Eulerian nor a tree, it follows from Theorems 15, 17 and 22 that $\mu/2 < v < \mu$. Yet, it may happen that the random Chinese postman tour is an optimal search strategy (as in the cases of Eulerian graphs and trees). In this section we analyze these types of graphs. In Section 4.1 we present a family of graphs for which the random Chinese postman tour is optimal and in Section 4.2 we present the widest family of graphs with this property.

4.1. Searching Weakly Cyclic Graphs

Definition 24 *A graph is called "weakly cyclic" if between any two points there exist at most two disjoint paths.*

An equivalent requirement, presented in [24] is that the graph has no subset topologically homeomorphic with a graph consisting of three arcs joining two points.

The difficulty in solving search games for the three arcs graph is presented in Section 5. Note that an Eulerian graph may be weakly cyclic (e.g. if all the nodes have degree 2) but need not be weakly cyclic (e.g., 4 arcs connecting two points).

It follows from the definition that if a weakly cyclic graph has a closed curve Γ with arcs $b_1, \ldots, b_k$ incident to it, then removing Γ disconnects Q into k disjoint graphs $Q_1, \ldots, Q_k$ with b_i belonging to Q_i. (If Q_i and Q_j would have been connected then the incidence points of Γ with b_i and with b_j could be connected by 3 disjoint paths). Thus, any weakly cyclic graph can be constructed by starting with a tree (which is obviously weakly cyclic) and replacing some of its nodes by closed curves as, for example, Γ and Γ_1 in Figure 4 (all edges have length 1). We leave the proof of this as an exercise to the reader.

We now formally define the above operation:

Definition 25 *"Shrinking a subgraph* Γ" : *Let a graph Q contain a connected subgraph Γ. If we replace the graph Q by a graph Q' in which Γ is replaced by a point B and all arcs in $Q \setminus \Gamma$ which are incident to Γ are incident to B in Q' we shall say that Γ was shrunk and B is the shrinking node.*

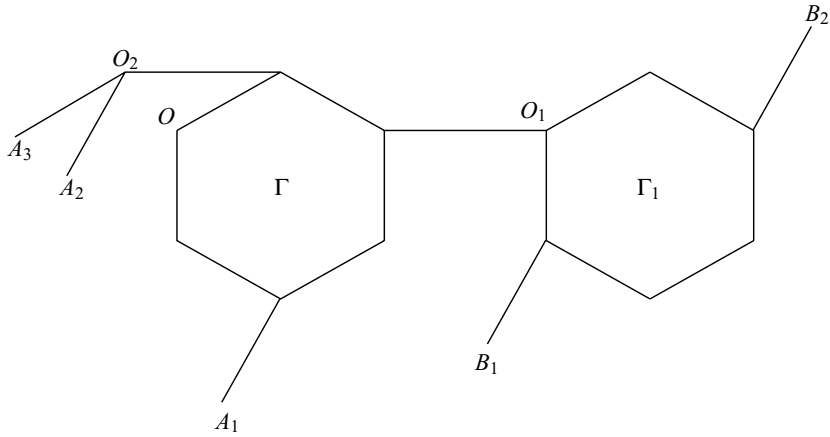


Figure 4.

It is easy to see that Q is a weakly cyclic graph if and only if it contains a set of *non-intersecting* curves $\Gamma_1, \ldots, \Gamma_k$ such that shrinking them transforms Q into a tree.

In order to obtain a feeling about optimal solutions for such graphs, we consider a simple example in which Q is a union of an interval of length l and a circle of circumference $\mu - l$ with only one point of intersection, as depicted in Figure 5. Assume, for the moment, that the searcher's starting point, O, is at the intersection.

Note that the length of the Chinese postman tour is $\bar{\mu} = \mu + l$. We now show that the value of the game satisfies $v = \frac{\mu+l}{2} = \frac{\bar{\mu}}{2}$ and the optimal search strategy, $\bar{s}$, is the random Chinese postman tour.

The random Chinese postman tour guarantees (at most) $\frac{\bar{\mu}}{2}$ by Lemma 13. The following target distribution, $\bar{h}$, guarantees (at least) $\frac{\bar{\mu}}{2}$: hide with probability $\bar{p} = \frac{2l}{\mu+l}$ at the end of the interval (at A) and with probability $1 - \bar{p}$ uniformly on the circle. (It can be easily checked that if the searcher either goes to A, returns to O and then goes around the circle, or encircles and later goes to A, then the expected capture time is equal to $\frac{\mu+l}{2}$. Also, any other search trajectory yields a larger expected capture time.)

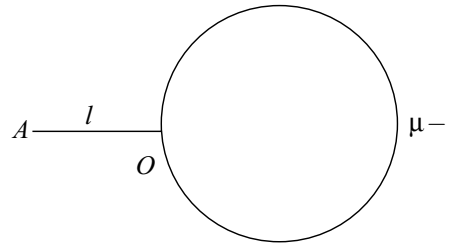


Figure 5.

Now assume that the starting point is different from O. In this case the value and the optimal search strategy remain the same, but the least favorable target hiding distribution is unchanged only if the starting point is (anywhere) on the circle. As the starting point moves from O to A the probability of hiding at A decreases from $\frac{2l}{\mu+l}$ to 0.

Solving the search games for weakly cyclic graphs was presented by Reijnierse and Potters [26]. In their paper they show that $v = \bar{\mu}/2$ and present an algorithm for constructing least favorable target distributions. (The optimal search strategy, as presented in Lemma 13, is the random Chinese postman tour.)

We now present a simpler version of their algorithm by transforming the graph Q into an "equivalent" tree $\bar{Q}$ as follows:

Algorithm 26 *Shrink each closed curve Γ_i, with circumference γ_i, and replace it by an arc c_i of length $\frac{\gamma_i}{2}$ which connects a (new) leaf C_i to the shrinking node B_i. All other arcs and nodes remain the same. Let $\bar{h}$ be the least favorable target distribution for the tree $\bar{Q}$. Then the least favorable target distribution for Q is obtained as follows:*

1. *For a leaf of Q (which is also a leaf of $\bar{Q}$) hide with the probability assigned to it by $\bar{h}$.*
2. *For a curve Γ_i (represented by leaf C_i in $\bar{Q}$) hide uniformly along it with overall probability assigned by $\bar{h}$ to leaf C_i.*

All other arcs and nodes are never chosen as hiding places.

We now use the above algorithm for the example presented in [26]. The graph Q is depicted in Figure 4 and its equivalent tree is depicted in Figure 2 (see Section 3). Note that the curves Γ and Γ_1 are replaced by arcs OC and O_1C_1. Thus, the least favorable probability distribution is the same for the leaves of Q and (replacing the leaves C and C_1 by Γ and Γ_1) hiding, uniformly, on Γ with probability $\frac{3}{13}$ (i.e. probability density $\frac{3}{78}$) and on Γ_1 with probability $\frac{18}{65}$ (i.e. probability density $\frac{18}{390}$).

4.2. Searching Weakly Eulerian Graphs

Reijnierse and Potters [26] conjectured that their algorithm (for the weakly cyclic graphs) of constructing the least favorable distribution and the result $v = \bar{\mu}/2$ hold for the wider family of *weakly Eulerian* graphs, i.e., graphs obtained from a tree replacing some nodes by Eulerian graphs. This conjecture was shown to be correct by Reijnierse [28].

They also conjectured that $v = \bar{\mu}/2$ implies that the graph is weakly Eulerian.

We provide in [13] a simple proof for the first conjecture and also show that their second conjecture is correct. In order to present our results we first formally define the graphs in question.

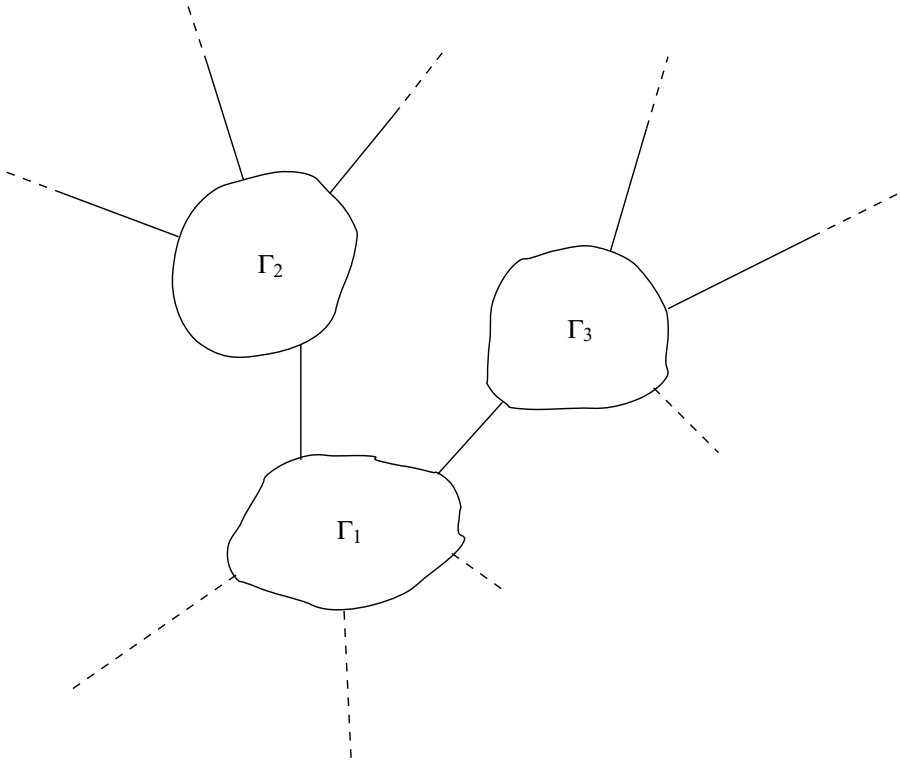


Figure 6.

Definition 27 *A graph is called "weakly Eulerian" if it contains a set of non-intersecting Eulerian graphs* $\Gamma_1, ..., \Gamma_k$ *such that shrinking them transforms* Q *into a tree.*

An equivalent condition is that removing all the arcs which disconnect the graph (the 'tree part') leaves a subgraph(s) with all nodes having an even (possibly zero) degree (see Figure 6).

Obviously, any Eulerian graph is also weakly-Eulerian.

Theorem 28 *If* Q *is weakly Eulerian, then:*

1. $v = \bar{\mu}/2$.
2. *A random Chinese postman tour is an optimal search strategy.*
3. *A least favorable hiding distribution,* $\bar{h}$, *is the distribution (similarly to Algorithm* 26) *for the equivalent tree,* $\bar{Q}$, *obtained by shrinking all the Eulerian subgraphs and replacing each such subgraph by an arc from the shrinking node to a (new) leaf with half the measure of the corresponding Eulerian subgraph.*

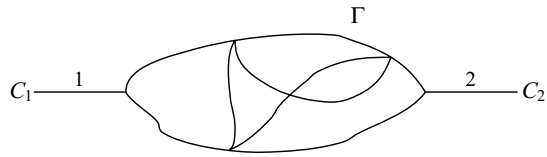


Figure 7.

Then, the probability of such a leaf is uniformly spread on the corresponding Eulerian subgraph. (The hiding probabilities at the leaves of Q remain the same.)

Proof. (A sketch.) Let the measure of the equivalent tree $\bar{Q}$ be $\bar{\mu}/2$. It can easily be verified that the length of a Chinese postman tour is $\bar{\mu}$ because each arc in the tree part of Q has to be traversed twice while the Eulerian subgraphs can be traversed just once. Thus, the searcher can achieve $\bar{\mu}/2$.

In [13] and [1] it is proven that $\bar{h}$ guarantees at least $\bar{\mu}/2$ expected search time, using an equivalent tree construction ■

(The probability of hiding in the Eulerian subgraphs is unique. For example, hiding in the middle of each arc, with the appropriate probability, is also optimal.)

We now illustrate the result by an example: Let Q be the union of an Eulerian graph Γ, of measure γ, and two arcs, of lengths 1 and 2 respectively, leading to leafs C_1 and C_2 (see Figure 7).

If $O \in \Gamma$ then, $\bar{Q}$ would be a star with three rays of lengths 1, 2, and 0.5γ, respectively. Thus, $\bar{h}$ hides at C_1 with probability $\frac{1}{0.5\gamma+3}$ at C_2 with probability $\frac{2}{0.5\gamma+3}$ and uniformly on Γ with overall probability $\frac{0.5\gamma}{0.5\gamma+3}$. If the starting point is on the arc leading to C_2 with distance 1 from C_2 then $\bar{Q}$ would be the tree depicted in Figure 8. Thus, the corresponding least favorable probabilities for C_1, C_2 and Γ would be $\frac{0.5\gamma+2}{0.5\gamma+3} \times \frac{1}{0.5\gamma+1}$, $\frac{1}{0.5\gamma+3}$ and $\frac{0.5\gamma+2}{0.5\gamma+3} \times \frac{0.5\gamma}{0.5\gamma+1}$, respectively.

Actually, the property $v = \bar{\mu}/2$ holds only for weakly Eulerian graphs:

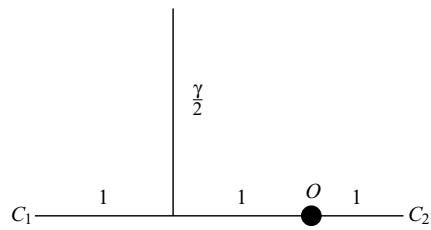


Figure 8.

Theorem 29 *Let Q be any graph. If $v = \bar{\mu}/2$ then Q is weakly Eulerian.*

Proof. (A sketch.) Let L be a Chinese postman tour (with length $\bar{\mu}$). Then the random Chinese postman tour of L is an optimal search strategy (since it guarantees $\bar{\mu}/2$). The tour L traverses some of the arcs once - call the set of such arcs T_1, and some arcs twice - call this set T_2.

It has to be shown that removing each arc of T_2 disconnects the graph. Since all the nodes in the set T_1 must have even degrees implying that T_1 is either an Eulerian subgraph or a set of Eulerian subgraphs, it would then follow that Q satisfies the condition equivalent to Definition 27.

Since $v = \bar{\mu}/2$, then for any $\varepsilon > 0$ there exists h_ε , a target distribution which guarantees at least $\bar{\mu}/2 - \varepsilon$ expected search time. Let $b \in T_2$ with length $l(b) > 0$. It can be shown that assuming $Q \setminus b$ is connected enables the searcher to use search strategies with expected search time less than $(\bar{\mu} - l(b))/2$ which would contradict the ε- optimality of h_ε. Thus, removing b has to disconnect Q. ■

Combining Theorems 28, 29 and 15 yields the following result:

Theorem 30 *For any graph Q, a random Chinese postman tour is an optimal search strategy if and only if the graph Q is weakly Eulerian.*

Conjecture 31 *Note that the value is $\bar{\mu}/2$, and the optimal search strategy is a random Chinese postman tour, independently of the specific starting point. This property is unique to the weakly Eulerian graphs.*

We conjecture that the independence of the value on the starting point holds only for weakly Eulerian graphs. (It does not hold in general, see Remark 34.)

Searching a graph which is not weakly Eulerian is expected to lead to rather complicated optimal search strategies. We shall show in the next section that even the "simple" graph with two nodes connected by three arcs, of unit length each, requires a mixture of infinitely many trajectories for the optimal search strategy.

5. Simple Graphs Requiring Complicated Search Strategies

In the previous section, we found optimal search strategies for weakly Eulerian graphs,. (which includes Eulerian graphs and the trees). We have shown that the random Chinese postman tour is optimal, and $v = \bar{\mu}/2$, only for this family. We now present a simple graph which is not weakly Eulerian and hence has value strictly less than $\bar{\mu}/2$.

In this example, the graph Q consists of k distinct arcs, $b_1, \ldots, b_k$, of unit length, connecting two points O and A. If the number k of arcs is even, then the graph Q is

Eulerian and the solution of the game is simple. On the other hand, it turns out that the solution is surprisingly complicated in the case that k is an odd number greater than 1, even if k is equal only to 3. For this graph $\bar{\mu} = k + 1$, implying from the last section that $v < \frac{k+1}{2}$. Actually, the following (stricter) inequality holds:

Lemma 32 *If Q is a set of k non-intersecting arcs of unit length which join O and A, and k is an odd number greater than 1, then*

$$v < \frac{k}{2} + \frac{1}{2k}. \tag{4}$$

Proof. (A sketch.) We use a natural symmetric search strategy $\bar{s}$: Starting from O make an equiprobable choice among the k arcs, (i.e., choosing each arc with probability $1/k$) and move along the chosen arc to A; then make an equiprobable choice among the $k-1$ remaining arcs, independently of the previous choice, and move along this arc back to O; then move back to A; and so on until all the arcs have been visited.

A simple calculation shows that if the distance between the target and A is d, then the expected search time will be

$$E(T) = \frac{k}{2} + \frac{1}{2k} - \frac{d}{k} < \frac{k}{2} + \frac{1}{2k}. \qquad \blacksquare$$

The case where Q consists of an odd number of arcs which connect two points has been used as an example for situations in which $v < \bar{\mu}/2$, but this case is interesting by itself. It is amazing that the optimal search strategy is simple for any even k (and also for any odd or even k if the hider is mobile, see [1]) but it is quite complicated even for the case that k is equal only to 3. The reasonable symmetric search strategy $\bar{s}$, used in proving Lemma 32, which is optimal for an even k, can assure the searcher an expected search time less than $\frac{k}{2} + \frac{1}{2k}$. However, if k is odd then the search strategy $\bar{s}$ is not optimal, or even ε-optimal, as can be easily deduced from the following argument. If the searcher uses $\bar{s}$, then a target distribution, with an expected search time close to $\frac{k}{2} + \frac{1}{2k}$, has to be concentrated near A with probability $\geq 1 - \varepsilon$. However, it can easily be verified that the expected search time guaranteed by such a hiding distribution does not exceed $1 + \delta$, where δ is small. Thus, the value of the search game has to be smaller than $\frac{k}{2} + \frac{1}{2k}$, which implies that the strategy $\bar{s}$ cannot be optimal, or even ε-optimal.

In order to demonstrate the complexity of this problem, we now consider the case $k = 3$. In this case, the symmetric search strategy $\bar{s}$ satisfies $v(\bar{s}) = 3/2 + 1/6 = 5/3$, but actually there exists a strategy $\tilde{s}$ which satisfies

$$v(\tilde{s}) = (4 + \ln 2)/3 < 5/3.$$

The strategy $\tilde{s}$ is a specific choice among the following family $\{s_F\}$ of search strategies:

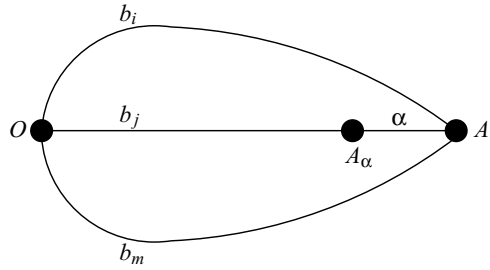


Figure 9.

Definition 33 *The family $\{s_F\}$ of search strategies is constructed as follows: Consider a set of trajectories $S_{ij\alpha}$, where i and j are two distinct integers in the set $\{1, 2, 3\}$ and $0 \le \alpha \le 1$. The trajectory S_{ij} starts from O, moves along b_i to A, moves along b_j to the point A_α which has a distance of α from A (see Figure 9), moves back to A, moves to O along b_m, where $m \in \{1, 2, 3\} \setminus \{i, j\}$, and then moves from O to A_α along b_j.*

Let $F(\alpha)$ be a cumulative probability distribution function of a random variable α $(0 \le \alpha \le 1)$. Then the strategy s_F is a probabilistic choice of a trajectory $S_{ij\alpha}$, where i is determined by an equiprobable choice in the set $\{1, 2, 3\}$, j is determined by an equiprobable choice in the set $\{1, 2, 3\} \setminus \{i\}$, and α is chosen independently, using the probability distribution F.

Note that the symmetric strategy $\bar{s}$ is a member of the family $\{s_F\}$ with the random variable α being identically zero.

The details of choosing the distribution function $\tilde{F}$ which corresponds to this strategy, $\tilde{s}$, is presented in [12] and [1].

Remark 34 *Unlike the weakly Eulerian graphs, the search value of the three-arcs graph depends on the starting point of the searcher. For example[2], assume that the searcher starts at the middle of arc b_1. Then, if the target distribution is h_μ (the uniform distribution) then the expected search time $\ge \frac{19}{12} > \frac{4+\ln 2}{3}$. (It can be verified that the searcher's best response is to go to one of the nodes, search b_2 and b_3, and finally retrace b_1 in order to search the unvisited half of b_1.)*

Problem 35 *Solving the search game in a graph with an arbitrary starting point for the searcher, is an interesting problem which has not been investigated yet.*
Let $\tilde{v}$ be the value of such a game. It is easy to see that $\tilde{v} \le \tilde{\mu}/2$, where $\tilde{\mu}$ is the minimum length of a path (not necessarily closed) which visits all the points of Q. Indeed, sometimes $\tilde{v} = \tilde{\mu}/2$ as happens for graphs with $\tilde{\mu} = \mu$, because the hider can guarantee expected search time $\ge \tilde{\mu}/2$ by using the uniform strategy h_μ. (Such an example is the 3 arcs search which is much more tractable with an arbitrary starting point than under the usual assumption of a fixed starting point known to the hider.)

[2]This observation was made by Steve Alpern

However, it is not clear for which family of graphs the equality $\tilde{v} = \tilde{\mu}/2$ *holds. For example, does it hold for trees?*

Remark 36 *The problem of finding the optimal search strategy for a (general) graph has been shown by von Stengel and Werchner [31] to be NP-hard. However, they also showed that if the search time is limited by a (fixed) bound which is logarithmic in the number of nodes, then the optimal strategy can be found in polynomial time.*

Finding an optimal search strategy on a graph can, in general, be formulated as an infinite-dimensional linear program. This formulation and an algorithm for obtaining its (approximate) solution is presented by Anderson and Armendia [2].

Conjecture 37 *Optimal (minimax) strategies for searching a target on any graph never use trajectories which visit some arcs (or parts of arcs) more than twice.*

(Note that this property does not hold against all target distributions. For example in the three arcs graph, if the hiding probability is $\frac{1}{3} - \varepsilon$ *for each point* $B_i, i = 1, 2, 3$, *having distance* $\frac{1}{4}$ *from* A, *and* ε *for* $C_i, i = 1, 2, 3$, *having distance* $\frac{1}{3}$ *from* A. *Then for a small* ε *the optimal search strategy is to go to* A, *then to one of the unvisited* B_i, *then through* A, *to the unvisited* B_j, *continue to* C_j *and finally from* C_j *to* C_i *through* B_i. *Thus, the segment* AB_i *is traversed* 3 *times. Still, we conjecture that such a situation cannot occur against a least favorable target distribution.)*

6. Search in an Unknown Graph

In the previous sections we examined the problem of searching for a target in a finite connected network Q of total length μ. We now make the searcher's problem more difficult by depriving him of a view of the whole network Q and instead let him see (and remember) only that part of Q which he has already traversed. We also let him remember the number of untried arcs leading off each of the nodes he has visited. Under these informational conditions, the network Q is known as a *maze*. The starting point O is known as the *entrance*, and the position A of the hider is known as the *exit*. In this section we present the randomized algorithm of [14] for minimizing the expected time for the searcher to find the exit in the worst case, relative to the choice of the network and the positioning of its entrance and exit. This strategy (called the randomized Tarry algorithm) may be interpreted as an optimal search strategy in a game in which the maze (network with entrance and exit) is chosen by a player (hider) who wishes to maximize the time required to reach the exit.

Finding a way for exiting a maze has been a challenging problem since ancient times. Deterministic methods which assure finding the exit were already known in the 19th century. Lucas (1882) described such an algorithm developed by [30] presented an algorithm which is now very useful in computer science for what is known as Depth-First Search. An attractive description of Depth-First Search is presented in Chapter 3 of [9]. The algorithms of Trémaux and Tarry mentioned above each guarantee that the searcher will reach the exit by time 2μ. [10, 11]

presented a variant of Tarry's algorithm which has an improved performance for some cases but has the same worst-case performance of 2μ.

The worst-case performance of any fixed search strategy cannot be less than 2μ. This can easily be seen by considering the family of "star" mazes consisting of n rays of equal length μ/n, all radiating from a common origin O. The least time required to visit all the nodes of this maze is $(n-1)(2\mu/n)+(\mu/n)=2\mu-\mu/n$, which converges to 2μ. Can this worst case performance be improved by using mixed strategies and requiring only that the *expected* time to find the exit is small? This question is related to the following game: For a given parameter μ, Player II (the hider) chooses a maze with measure μ and specifies the entrance O and the exit A. Player I (the searcher) starts from O and moves at unit speed until the first time T (the payoff to the maximizing player II) that he reaches A. We will show that this game has a value v which is equal to the measure μ of the maze. Obviously $v \geq \mu$, because the hider can always choose the maze to be a single arc of length μ going from O to A. On the other hand, we shall show that the searcher has a mixed strategy which guarantees that the expected value of T does not exceed μ. Consequently $v=\mu$. Therefore, this (optimal) search strategy achieves the best worst-case performance for reaching the exit of a maze. This result was obtained by Gal and Anderson [14] on which the following discussion is based.

The optimal strategy generates random trajectories which go from node to node by traversing the intervening arc at unit speed without reversing direction. Consequently it is completely specified by giving (random) rules for leaving any node that it reaches. This strategy is based in part on a coloring algorithm on the *passages* of the maze. A passage is determined by a node and an incident arc. Thus each arc $a=BC$ has two passages, one at each end, the "leaving" passage (B, a) and the "arriving" passage (C, a). We assume that initially the passages are uncolored, but when we go through a passage we may sometimes color it either in yellow or in red. The strategy has two components: *coloring rules* and *path rules* as follows. Since this strategy is in fact a randomized version of Tarry's algorithm, we will refer to it as the *randomized Tarry algorithm*.

7. Randomized Tarry Algorithm

7.1. Coloring rules

1. When arriving at any node *for the first time*, color the arriving passage in *yellow.* (This will imply that there is at most one yellow passage at any node.)
2. When leaving any node, color the leaving passage in *red.*

7.2. Path rules (how to leave a node)

1. If there are uncolored passages, choose among them equiprobably.
2. If there are no uncolored passages but there is a yellow passage choose it for leaving the node, thus changing its color to red.
3. If there are only red passages, stop.

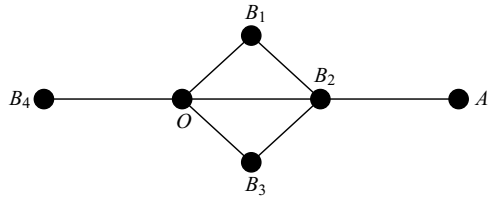


Figure 10.

Obviously when adopting the above algorithm to exit a maze, the searcher would use the obvious additional rule of stopping when the exit A is reached. However, for analytical reasons we prefer to consider the full paths $\tau(t)$ produced when the searcher does not stop at A, but continues until the single stopping rule 3 is applied. We call these full paths *tours* and denote the set of all tours by $\mathcal{T}$. The set $\mathcal{T}$ is finite because there are only finitely many randomizations involved in the generation of the tours.

To illustrate how the randomized Tarry algorithm generates tours of the maze, consider the particular maze drawn in Figure 10.

All arcs have length 1 so that $\mu = 7$. For this maze a particular tour $\bar{\tau} \in \mathcal{T}$ is given by the node sequence (which determines the continuous path)

$$\bar{\tau} = O, B_1, B_2, O, B_2, B_3, O, B_4, O, B_3, B_2, A, B_2, B_1, O. \tag{5}$$

The tour starts by coloring (O, OB_1) red, (B_1, OB_1) yellow, (B_1, B_1B_2) red, (B_2, B_1B_2) yellow and (B_2, OB_2) red. It then continues coloring (O, OB_2) red, (B_2, B_2B_3) red, (B_3, B_2B_3) yellow, (B_3, B_3O) red, etc. Note that the tour $\bar{\tau}$ stops at O when all its passages become red. It traverses each arc exactly once in each direction and consequently takes 14 $(= 2\mu)$ time units. The following Lemma shows that this behavior is not specific to the maze or the particular tour chosen but *always* occurs.

Lemma 38 *Let τ be any tour generated by the randomized Tarry algorithm for a particular maze Q of total length μ and entrance O. Then τ finishes at O at time 2μ, after traversing every arc of Q exactly once in each direction.*

Proof. Note that such a realization uses Tarry's algorithm for visiting all arcs. Thus, Lemma 38 is simply a reformulation of Tarry's theorem (see Theorem 1 of [10]). ■

Since each tour τ visits all points of the maze before returning to the entrance at time 2μ, an immediate consequence is the following.

Corollary 39 *The randomized Tarry algorithm guarantees that the time to reach the exit is smaller than 2μ.*

We have already observed that the set of tours is finite. The following lemma computes its cardinality, which is also the reciprocal of the probability of any particular tour.

Lemma 40 *Let Q be a maze with $I+1$ nodes. Denote by d_0 the degree of O and by d_i the degree of the remaining nodes i, $i=1,\ldots,I$. Then the cardinality K of the set $\mathcal{T}$ consisting of all tours τ is given by*

$$K = \#\mathcal{T} = d_0! \prod_{i=1}^{I} (d_i - 1)!.$$

Each tour τ is obtained by the randomized Tarry algorithm with an equal probability of $1/K$.

The proof is given in [14] and in [1].

We now consider the main problem of evaluating the expected time taken for the randomized Tarry algorithm to reach a point A of the maze (which might be taken as the exit). For $\tau \in \mathcal{T}$ and any node A of Q, define $T_A(\tau)$ to be the first time that the tour τ reaches the node A, that is $T_A(\tau) = \min\{t : \tau(t) = A\}$. Consequently the *expected* time $\tilde{T}_A$ for the randomized Tarry algorithm to reach A is given by

$$\tilde{T}_A = \frac{1}{K} \sum_{\tau \in \mathcal{T}} T_A(\tau). \tag{6}$$

In order to obtain an upper bound on $\tilde{T}_A$ it will be useful to compare the performance of a tour τ with the performance of its *reverse tour* $\phi(\tau) = \tau'$ defined by

$$[\phi(\tau)](t) \equiv \tau'(t) = \tau(2\mu - t). \tag{7}$$

For example, the reverse tour of the tour $\bar{\tau}$ given in (5) is expressed in node sequence form as

$$\bar{\tau}' = O, B_1, B_2, A, B_2, B_3, O, B_4, O, B_3, B_2, O, B_2, B_1, O.$$

Observe that while the time $T_A(\bar{\tau}) = 11$ is greater than $\mu = 7$, when averaged with the smaller time of $T_A(\bar{\tau}') = 3$ for its reverse path, we get $(11+3)/2 = 7$. This motivates the following analysis. Consequently we can just as well calculate $\tilde{T}_A$ by the formula

$$\tilde{T}_A = \frac{1}{K} \sum_{\tau \in \mathcal{T}} T_A(\tau'), \text{ and hence also by}$$

$$\tilde{T}_A = \frac{1}{K} \sum_{\tau \in \mathcal{T}} \frac{T_A(\tau) + T_A(\tau')}{2}. \tag{8}$$

Observe that since $\tau(T_A(\tau)) = A$ by definition, we have by (7) that $\tau'[2\mu - T_A(\tau)] = A$.

If τ reaches A exactly once (which is the same as A having degree 1) then so does τ' and consequently $T_A(\tau') = 2\mu - T_A(\tau)$. If τ also reaches A at some later time, then τ' also reaches A at some earlier time $(2\mu - \max\{t : \tau(t) = A\})$, and hence $T_A(\tau') < 2\mu - T_A(\tau)$. In either case, we have that

$$T_A(\tau') \leq 2\mu - T_A(\tau)$$

i.e.,

$$T_A(\tau') + T_A(\tau) \leq 2\mu, \tag{9}$$

with equality if and only if A is a node of degree 1. Combining (8) and (9), we have

$$\begin{aligned}\tilde{T}_A &= \frac{1}{K}\sum_{\tau\in\mathcal{T}}\frac{T_A(\tau) + T_A(\tau')}{2} \\ &\leq \frac{1}{K}\sum_{\tau\in\mathcal{T}}\frac{2\mu}{2} = \mu.\end{aligned}$$

Again, we have equality if and only if A has degree 1. Since in this argument A was arbitrary, we have established the following.

Theorem 41 *The randomized Tarry algorithm reaches every possible exit point in Q in expected time not exceeding the total length μ of Q. Furthermore, except for nodes of degree 1, it reaches all points in expected time strictly less than μ.*

Since we have already observed that the hider can choose a maze in which the exit lies at the end of a single arc of length μ (which would take time μ to reach) we have the following.

Corollary 42 *The value of the game in which the hider chooses a maze of total length μ, with entrance and exit, and the searcher moves to minimize the (Payoff) time required to find the exit, is equal to μ.*

We conclude this section by giving some comparisons of the randomized Tarry algorithm with other possible methods of searching a maze. Notice that in the Tarry algorithm a tour (such as $\bar{\tau}$ given in (5) on the first return to O) may choose an arc that has already been traversed (in the opposite direction), even when an untraversed arc is available at that node. This may appear to be a foolish choice because the untraversed node may lead to a new part of the maze (maybe even straight to the exit). Yet, a strategy which is based on giving priority to unvisited arcs leads to significantly inferior performance in the worst case: 2μ instead of μ, as shown in [14].

The randomized Tarry algorithm requires only local information in each node. This type of locally determined strategy is similar to some policies that could be employed in a distributed computing environment to deal with incoming messages or queries directed to an unknown node of the computer network. Such a model was used, for example, by Golumbic [15]. Our strategy provides an alternative path finding mechanism which uses only local information. Other local information schemes of searching an unknown graph by several (decentralized) agents (ant-robots which leave chemical odor traces) are described and analyzed by Wagner, Lindenbaum and Bruckstein [32–34].

8. Further Reading

Searching strategies for graphs include many other fascinating topics which could not be covered in this survey. We mention some of them here, and encourage interested readers to read also [1] which cover these topics.

- *Searching for a Moving Target.* Except for the circle and some other specific examples finding optimal search strategies is still open and seem difficult, even for simple graphs like trees.
- *Searching in a Graph with Infinite Length.* This includes the well known Linear Search Problem of finding an optimal search trajectory on the infinite line and some of its extensions. The optimal search trajectories given by the "turning points" usually form geometric sequences.
- *Searching for a Target who Wants to be Found.* Assume, for example, that two searchers randomly placed in a graph wish to minimize the time required to meet. Rendezvous problems of this type include several variants. In a relatively simple case the searchers have a common labeling of the graph so they can agree to meet at an agreed point. The problem becomes more difficult if such a common labeling does not exist. For example, finding optimal strategies for meeting on the line is quite a difficult problem.

Acknowledgments

The author is grateful to Steve Alpern, Irith Ben-Arroyo Hartman, and Martin Golumbic for their help in preparing this article. The author was supported, in part, by NATO Grant PST.CLG.976391.

References

[1] Alpern, S.; Gal, S. Search Games and Rendezvous Theory. Kluwer academic publishers (2003).

[2] Anderson, E. J.; Aramendia, M. A. The search game on a network with immobile hider. *Networks* 20: 817–844 (1990).

[3] Benkoski, S.J., Monticono, M.G., and Weisinger, J.R. A survey of the search theory literature, *Naval Res. Log.* 38: 469–494 (1991).

[4] Berge, C. Graphs and Hypergraphs. North-Holland, Publ Amsterdam (1973).

[5] Christofides, N. Graph Theory: An Algorithmic Approach. Academic Press, New York (1975).

[6] Edmonds, J. The Chinese postman problem. *Bull. Oper. Res. Soc. Amer.* 13, Suppl. 1, B–73 (1965).

[7] Edmonds, J., and Johnson, E. L. Matching Euler tours and the Chinese postman problem. *Math. Programming* 5: 88–124 (1973).

[8] Eiselt, H. A., Gendreau, M., and Laporte, G. Arc routing problems. I. The Chinese postman problem. *Oper. Res.* 43: 231–242 (1995).

[9] Even, S. Graph Algorithms, Computer Science Press, Rockville, MD (1979).

[10] Fraenkel, A. S. Economic traversal of labyrinths. *Mathematics Magazine* 43: 125–130 (1970).

[11] Fraenkel, A. S. Economic traversal of labyrinths. *Mathematics Magazine* 44: 12 (1971).

[12] Gal, S. Search Games, Academic Press, New York (1980).

[13] Gal, S. On the optimality of a simple strategy for searching graphs, *Internat. J. Game Theory* 29: 533–542 (2000).

[14] Gal, S. and Anderson, E. J. Search in a maze, *Probability in the Engineering and Informational Sciences* 4: 311–318 (1990).

[15] Golumbic, M. C. A general method for avoiding cycling in networks, *Information Processing Letters* 24: 251–253 (1987).

[16] Kikuta, K. A hide and seek game with traveling cost. *J. Oper. Res. Soc. Japan* 33: 168–187 (1990).

[17] Kikuta, K. A search game with traveling cost. *J. Oper. Res. Soc. Japan* 34: 365–382 (1991).

[18] Kikuta, K. A search game with traveling cost on a tree. *J. Oper. Res. Soc. Japan* 38: 70–88 (1995).

[19] Kikuta, K. and Ruckle, W. H. Initial point search on weighted trees. *Naval Res. Logist.* 41: 821–831 (1994).

[20] Koopman, B. O. Search and Screening: General Principles with Historical Applications, Pergamon Press, New York, (1980).

[21] Lawler, E. L. Combinatorial optimization: Networks and Matroids. Holt, New York (1976).

[22] Megiddo, N., and Hakimi, S. L. Pursuing mobile hider in a graph. The Center for Math. Studies in Econom. and Management Sci., Northwestern Univ., Evanston, Illinois, Disc. paper No. 360, 25 (1978).

[23] Megiddo, N.; Hakimi, S. L.; Garey, M. R.; Johnson, D. S.; Papadimitriou, C. H. The complexity of searching a graph. *J. Assoc. Comput. Mach.* 35: 18–44 (1988).

[24] Parsons, T. D. Pursuit-evasion in a graph. In Theory and Application of Graphs (Y. Alavi and P. R. Lick, eds.). Springer-Verlag, Berlin (1978a).

[25] Parsons, T. D. The search number of a connected graph. Proc. Southwestern Conf. Combinatorics, Graph Theory, and Computing, 9th, Boca Raton, Florida (1978b).

[26] Reijnierse, J. H. and Potters J. A. M. Search Games with Immobile Hider, *Internat. J. Game Theory* 21: 385–394 (1993a).

[27] Reijnierse J. H. and Potters J. A. M. Private communication (1993b).

[28] Reijnierse JH. Games, graphs and algorithms, Ph. D Thesis, University of Nijmegen, The Netherlands (1995).

[29] Stone, L. D. Theory of Optimal Search. 2nd ed., Operations Research Society of America, Arlington VA (1989).

[30] Tarry, G. La problem des labyrinths. *Nouvelles Annales de Mathematiques* 14: 187 (1895).

[31] Von Stengel, B. and Werchner, R. Complexity of searching an immobile hider in a graph. *Discrete Appl. Math.* 78: 235–249 (1997).

[32] Wagner, Israel A.; Lindenbaum, Michael; Bruckstein, Alfred M. Smell as a computational resource—a lesson we can learn from the ant. Israel Symposium on Theory of Computing and Systems (Jerusalem, 1996), 219–230, IEEE Comput. Soc. Press, Los Alamitos, CA (1996).

[33] Wagner, Israel A.; Lindenbaum, Michael; Bruckstein, Alfred M. Efficiently searching a graph by a smell-oriented vertex process. Artificial intelligence and mathematics, VIII (Fort Lauderdale, FL, 1998). *Ann. Math. Artificial Intelligence* 24: 211–223 (1998).

[34] Wagner, I. A., Lindenbaum, M., and Bruckstein, A. M. Ants: Agents on networks, trees, and subgraphs, *Future Generation Computer Systems* 16: 915–926 (2000).

9

Recent Trends in Arc Routing

Alain Hertz

Ecole Polytechnique - GERAD
Département de Mathématiques et de génie industriel
CP 6079, succ. Centre-ville
Montréal (QC) H3C 3A7, Canada

Abstract

Arc routing problems (ARPs) arise naturally in several applications where streets require maintenance, or customers located along road must be serviced. The undirected rural postman problem (URPP) is to determine a least cost tour traversing at least once each edge that requires a service. When demands are put on the edges and this total demand must be covered by a fleet of identical vehicles of capacity Q based at a depot, one gets the undirected capacitated arc routing problem (UCARP). The URPP and UCARP are known to be NP-hard. This chapter reports on recent exact and heuristic algorithms for the URPP and UCARP.

1. Introduction

Arc routing problems (ARPs) arise naturally in several applications related to garbage collection, road gritting, mail delivery, network maintenance, snow clearing, etc. [19, 1, 14]. ARPs are defined over a graph $G = (V, E \cup A)$, where V is the vertex set, E is the edge set, and A is the arc set. A graph G is called *directed* if E is empty, *undirected* if A is empty, and *mixed* if both E and A are non-empty. In this chapter, we consider only undirected ARPs. The traversal *cost* (also called *length*) c_{ij} of an edge (v_i,v_j) in E is supposed to be non-negative. A *tour* T, or *cycle* in G is represented by a vector of the form $(v_1,v_2,\ldots,v_n)$ where (v_i,v_{i+1}) belongs to E for $i = 1,\ldots,n-1$ and $v_n = v_1$. All graph theoretical terms not defined here can be found in [2].

In the *Undirected Chinese Postman Problem*, one seeks a minimum cost tour that traverses all edges of E at least once. In many contexts, however, it is not necessary to *traverse* all edges of E, but to *service* or *cover* only a subset $R \subseteq E$ of *required* edges, traversing if necessary some edges of $E\backslash R$. A *covering tour* for R is a tour that traverses all edges of R at least once. When R is a proper subset of E, the problem of finding a minimum cost covering tour for R is known as the *Undirected Rural Postman*

Problem (URPP). Assume for example that a city's electric company periodically has to send electric meter readers to record the consumption of electricity by the different households for billing purposes. Suppose that the company has already decided who will read each household's meter. This means that each meter reader has to traverse a given subset of city streets. In order to plan the route of each meter reader, it is convenient and natural to represent the problem as a URPP in which the nodes of the graph are the street intersections while the edges of the graph are the street segments between intersections, some of them requiring meter readings.

Extensions of these classical problems are obtained by imposing capacity constraints. The *Undirected Capacitated Arc Routing Problem* (UCARP) is a generalization of the URPP in which m identical vehicles are available, each of capacity Q. One particular vertex is called the *depot* and each required edge has an integral non-negative *demand*. A vehicle route *is feasible* if it contains the depot and if the total demand on the edges covered by the vehicle does not exceed the capacity Q. The task is to find a set of m feasible vehicle routes of minimum cost such that each required edge is serviced by exactly one vehicle. The number m of vehicles may be given a priori or can be a decision variable. As an example, consider again the above problem of the city's electric company, but assume this time that the subset of streets that each meter reader has to visit is not fixed in advance. Moreover, assume that no meter reader can work more than a given number of hours. The problem to be solved is then a UCARP in which one has to build a route for each meter reader so that all household's meters are scanned while no meter reader is assigned a route which exceeds the specified number of work hours.

The URPP was introduced by Orloff [40] and shown to be NP-hard by Lenstra and Rinnooy Kan [35]. The UCARP is also NP-hard since the URPP reduces to it whenever Q is greater than or equal to the total demand on the required edges. Even finding a 0.5 approximation to the UCARP is NP-hard, as shown by Golden and Wong [27]. The purpose of this chapter is to survey some recent algorithmic developments for the URPP and UCARP. The algorithms described in this chapter should be considered as skeletons of more specialized algorithms to be designed for real life problems which typically have additional constraints. For example, it can be imposed that the edges must be serviced in an order that respects a given precedence relation [15]. Also, real life problems can have multiple depot locations [17] and time windows or time limits on the routes [18, 44]. Arc routing applications are described in details in chapters 10, 11 and 12 of the book edited by Dror [14].

In the next section, we give some additional notations, and we describe a reduction that will allow us to assume that all vertices are incident to at least one required edge. We also briefly describe some powerful general solution methods for integer programming problems. Section 3 contains recent exact methods for the URPP. Then in Section 4, we describe some recent basic procedures that can be used for the design of heuristic methods for the URPP and UCARP. Section 5 contains examples of recent effective algorithms that use the basic procedures of Section 4 as main ingredients.

2. Preliminaries

Let V_R denote the set of vertices incident to at least one edge in R. The *required subgraph* $G_R(V_R,R)$ is defined as the partial subgraph of G induced by R. It is obtained from G by removing all non-required edges as well as all vertices that are not incident to any required edge. Let C_i $(i = 1, \ldots, p)$ be the i-th connected component of $G_R(V_R,R)$, and let $V_i \subseteq V_R$ be the set of vertices of C_i. [9] have designed a pre-processing procedure which converts any URPP instance into another instance for which $V = V_R$, (i.e. each vertex is incident with at least one required edge). This is done as follows. An edge (v_i,v_j) is first included in $G_R(V_R,R)$ for each v_i,v_j in V_R, with cost c_{ij} equal to the length of a shortest chain between v_i and v_j in G. This set of new edges added to $G_R(V_R,R)$ is then reduced by eliminating

(a) all new edges (v_i,v_j) for which $c_{ij} = c_{ik} + c_{kj}$ for some v_k in V_R, and
(b) one of two parallel edges if they have the same cost.

To illustrate, consider the graph G shown in Figure 1(a), where edges of R are shown in bold lines and numbers correspond to edge costs. The new instance with $V = V_R$ is represented in Figure 1(b).

From now on we will assume that the URPP is defined on a graph $G = (V,E)$ to which the pre-processing procedure has already been applied.

To understand the developments of Section 3, the reader has to be familiar with basic concepts in *Integer Programming*. If needed, a good introduction to this topic can be found in [45]. We briefly describe here below the *cutting plane algorithm* and the *Branch & Cut* methodology which are currently the most powerful exact solution approaches for arc routing problems.

Most arc routing problems can be formulated in the form

$$\begin{cases} \text{Minimize} \sum_{e \in E} c_e x_e \\ \text{subject to} \quad x \in S \end{cases}$$

where S is a set of *feasible solutions*. The convex hull *conv(S)* of the vectors in S is a polyhedron with integral extreme points. Since any polyhedron can be described by a set of linear inequalities, one can theoretically solve the above problem by Linear

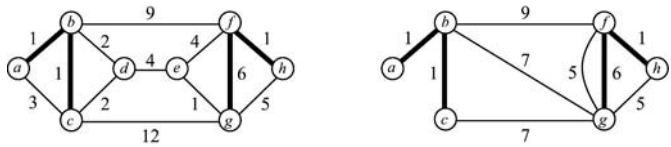


Figure 1. Illustration of the pre-processing procedure.

Programming (LP). Unfortunately, a complete linear description of *conv(S)* typically contains a number of inequalities which is exponential in the size of the original problem. To circumvent this problem, one can start the optimization process with a small subset of known inequalities and compute the optimal LP solution subject to these constraints. One can then try to identify an inequality that is valid for *conv(S)* but violated by the current LP solution. Such an inequality is called a *cutting plane*, because, geometrically speaking, it "cuts off " the current LP solution. If such a cutting plane is found, then it is added to the current LP and the process is repeated. Otherwise, the current LP solution is the optimal solution of the original problem. This kind of procedure is called the *cutting plane algorithm*. It originated in the pioneering work of Dantzig, Fulkerson and Johnson [12] on the Symmetric Traveling Salesman Problem.

The problem consisting in either finding a cutting plane, or proving that no such inequality exists is known as the *separation problem*. An algorithm that solves it is called an *exact separation algorithm*. The separation problem can however be NP-hard for some classes of inequalities. In such a case, one has to resort to a *heuristic separation algorithm* that may fail to find a violated inequality in the considered class, even if one exists.

The cutting plane algorithm stops when no more valid inequality can be found. This however does not mean that no such inequality exists. It may be that a violated inequality belongs to an unknown class of inequalities, or that it belongs to a known class for which we have used, without success, a heuristic separation problem. When the cutting plane algorithm fails to solve a given instance, one can choose among several options. One option is to feed the current LP solution into a classical Branch & Bound algorithm for integer programs. A more powerful option is to use the so-called *Branch & Cut* method (see for example [42]). A Branch & Cut is much like a Branch & Bound method except for the fact that valid inequalities may be added at any node of the branching tree. This leads to stronger linear relaxations at any node, which normally leads in turn to a considerable reduction in the number of nodes, in comparison with standard Branch & Bound.

3. Exact Algorithms for the URPP

A connected graph is said to be *Eulerian* if each vertex has an even degree. It is well known that finding a tour in an Eulerian graph that traverses each edge exactly once is an easy problem that can be solved, for example, by means of the $O(|E|)$ algorithm described by Edmonds and Johnson [16]. Hence, the URPP is equivalent to determining a least cost set of additional edges that, along with the required edges, makes up an Eulerian subgraph. Let x_e denote the number of copies of edge e that must be added to R in order to obtain an Eulerian graph, and let $G(x)$ denote the resulting Eulerian graph.

For a subset $W \subseteq V$ of vertices, we denote $\delta(W)$ the set of edges of E with one endpoint in W and the other in $V \setminus W$. If W contains only one vertex v, we simply write

$\delta(v)$ instead of $\delta(\{v\})$. [9] have proposed the following integer programming formulation for the URPP:

$$\text{Minimize} \sum_{e \in E} c_e x_e$$

subject to

$$|R \cap \delta(v_i)| + \sum_{e \in \delta(v_i)} x_e = 2z_i \qquad (v_i \in V) \tag{1}$$

$$\sum_{e \in \delta(W)} x_e \geq 2 \qquad (W = \bigcup_{k \in P} V_k,\ P \subset \{1, \ldots, p\},\ P \neq \emptyset) \tag{2}$$

$$x_e \geq 0 \text{ and integer} \qquad (e \in E) \tag{3}$$

$$z_i \geq 0 \text{ and integer} \qquad (v_i \in V) \tag{4}$$

Constraints (1) stipulate that each vertex in $G(x)$ must have an even degree. Indeed, the left-hand side of the equality is the total number of edges incident to v_i in $G(x)$, while the right-hand side is an even integer.

Constraints (2) enforce the solution to be connected. To understand this point, remember first that $G_R(V, R)$ contains p connected components with vertex sets $V_1, \ldots, V_p$. Now, let P be a non-empty proper subset of $\{1, \ldots, p\}$ and consider the vertex set $W = \bigcup_{k \in P} V_k$. Notice that no required edge has one endpoint in W and the other one outside W. In order to service not only the required edges with both endpoints in W, but also those with both endpoints in $V \setminus W$, a tour must traverse the frontier between W and $V \setminus W$ at least twice. Hence $\sum_{e \in \delta(W)} x_e$ must be at least equal to 2.

The associated polyhedron was not examined in detail by [9]. This was done in [10] who proposed the following formulation that avoids variables z_i and where $\delta_R(W) = R \cap \delta(W)$.

$$\text{Minimize} \sum_{e \in E} c_e x_e$$

subject to

$$\sum_{e \in \delta(v)} x_e = |\delta_R(v)| \,(\text{mod}\, 2) \qquad (v \in V) \tag{5}$$

$$\sum_{e \in \delta(W)} x_e \geq 2 \qquad (W = \bigcup_{k \in P} V_k,\ P \subset \{1, \ldots, p\},\ P \neq \emptyset) \tag{2}$$

$$x_e \geq 0 \text{ and integer} \qquad (e \in E) \tag{3}$$

The convex hull of feasible solutions to (2), (3), (5) is an unbounded polyhedron. The main difficulty with this formulation lies with the non-linear degree constraints (5).

Another approach has recently been proposed by Ghiani and Laporte [24]. They use the same formulation as Corberán and Sanchis, but they noted that only a small set of variables may be greater than 1 in an optimal solution of the RPP and, furthermore, these variables can take at most a value of 2. Then, by duplicating these latter variables, Ghiani and Laporte formulate the URPP using only 0/1 variables. More precisely, they base their developments on *dominance relations* which are equalities or inequalities that reduce the set of feasible solutions to a smaller set which surely contains an optimal solution. Hence, a dominance relation is satisfied by at least one optimal solution of the problem but not necessarily by all feasible solutions. While some of these domination relations are difficult to prove, they are easy to formulate. For example, [9] have proved the following domination relation.

Domination relation 1
Every optimal solution of the URPP satisfies the following relations:

$$x_e \leq 1 \quad \text{if } e \in R$$
$$x_e \leq 2 \quad \text{if } e \in E \setminus R$$

This domination relation indicates that given any optimal solution x^* of the URPP, all edges appear at most twice in $G(x^*)$. This means that one can restrict our attention to those feasible solutions obtained by adding at most one copy of each required edge, and at most two copies of each non-required one. The following second domination relation was proved by Corberán and Sanchis [10].

Domination relation 2
Every optimal solution of the URPP satisfies the following relation:

$x_e \leq 1$ if e is an edge linking two vertices in the same connected component of G_R

The above domination relation not only states (as the first one) that it is not necessary to add more that one copy of each required edge (i.e., $x_e \leq 1$ if $e \in R$), but also that it is not necessary to add more than one copy of each non-required edge linking two vertices in the same connected component of G_R. Another domination relation is given in [24].

Domination relation 3
Let G^* be an auxiliary graph having a vertex w_i for each connected component C_i of G_R and, for each pair of components C_i and C_j, an edge (w_i,w_j) corresponding to a least cost edge between C_i and C_j. Every optimal solution of the URPP satisfies the following relation:

$x_e \leq 1$ if e does not belong to a minimum spanning tree on G^*.

Let E_2 denote the set of edges belonging to a minimum spanning tree on G^*, and let $E_1 = E \setminus E_2$. The above relation, combined with domination relation 1 proves

that given any optimal solution x^* of the URPP, the graph $G(x^*)$ is obtained from G_R by adding at most one copy of each edge in E_1, and at most two copies of each edge in E_2. In summary, every optimal solution of the URPP satisfies the following relations:

$$x_e \leq 1 \quad \text{if} \quad e \in E_1$$
$$x_e \leq 2 \quad \text{if} \quad e \in E_2$$

Ghiani and Laporte propose to replace each edge $e \in E_2$ by two parallel edges e' and e''. Doing this, they replace variable x_e that can take values 0, 1 and 2 by two binary variables $x_{e'}$ and $x_{e''}$. Let E' and E'' be the set of edges e' and e'' and let $E^* = E_1 \cup E' \cup E''$. The URPP can now be formulated as a binary integer program:

$$\text{Minimize} \sum_{e \in E^*} c_e x_e$$

subject to

$$\sum_{e \in \delta(v)} x_e = |\delta_R(v)| \pmod 2 \qquad (v \in V) \tag{5}$$

$$\sum_{e \in \delta(W)} x_e \geq 2 \qquad (W = \bigcup_{k \in P} V_k, P \subset \{1, \ldots, p\}, P \neq \emptyset) \tag{2}$$

$$x_e = 0 \text{ or } 1 \qquad (e \in E^*) \tag{6}$$

The convex hull of feasible solutions to (2), (5), (6) is a polytope (i.e., a bounded polyhedron). The *cocircuits inequalities*, defined by Barahona and Grötschel [3] and described here below, are valid inequalities for this new formulation, while they are not valid for the unbounded polyhedron induced by the previous formulations. These inequalities can be written as follows:

$$\sum_{e \in \delta(v) \setminus F} x_e \geq \sum_{e \in F} x_e - |F| + 1 \qquad (v \in V,\ F \subseteq \delta(v), |\delta_R(v)| + |F| \text{ is odd}) \tag{7}$$

To understand these inequalities, consider any vertex v and any subset $F \subseteq \delta(v)$ of edges incident to v, and assume first that there is at least one edge $e \in F$ with $x_e = 0$ (i.e., no copy of e is added to $G_R(V,R)$ to obtain $G(x)$). Then $\sum_{e \in F} x_e - |F| + 1 \leq 0$ and constraints (7) are useless in that case since we already know from constraints (6) that $\sum_{e \in \delta(v) \setminus F} x_e$ must be greater than or equal to zero. So suppose now that $G(x)$ contains a copy of each edge $e \in F$. Then vertex v is incident in $G(x)$ to $|\delta_R(v)|$ required edges and to $|F|$ copies of edges added to $G_R(V,R)$. If $|\delta_R(v)| + |F|$ is odd then at least one additional edge in $\delta(v) \setminus F$ must be added to $G_R(V,R)$ in order to get the desired Eulerian graph $G(x)$. This is exactly what is required by constraints (7) since, in that case, $\sum_{e \in F} x_e - |F| + 1 = 1$. Ghiani and Laporte have shown that the non-linear

constraints (5) can be replaced by the linear constraints (7), and they therefore propose the following binary linear formulation to the URPP:

$$\text{Minimize} \sum_{e \in E^*} c_e x_e$$

subject to

$$\sum_{e \in \delta(v) \setminus F} x_e \geq \sum_{e \in F} x_e - |F| + 1 \qquad (v \in V,\ F \subseteq \delta(v), |\delta_R(v)| + |F| \text{ is odd}) \quad (7)$$

$$\sum_{e \in \delta(W)} x_e \geq 2 \qquad (W = \bigcup_{k \in P} V_k,\ P \subset \{1, \ldots, p\},\ P \neq \emptyset) \quad (2)$$

$$x_e = 0 \text{ or } 1 \qquad (e \in E^*) \quad (6)$$

All constraints in the above formulation are linear, and this makes the use of Branch & Cut algorithms easier (see Section 2). Cocircuit inequalities (7) can be generalized to any non-empty subset W of V:

$$\sum_{e \in \delta(W) \setminus F} x_e \geq \sum_{e \in F} x_e - |F| + 1 \qquad (F \subseteq \delta(W), |\delta_R(W)| + |F| \text{ is odd}) \quad (8)$$

If $\delta_R(W)$ is odd and F is empty, constraints (8) reduce to the following *R-odd inequalities* used by Corberán and Sanchis [10]:

$$\sum_{e \in \delta(W)} x_e \geq 1 \qquad (W \subset V, |\delta_R(W)| \text{ is odd}) \quad (9)$$

If $\delta_R(W)$ is even and F contains one edge, constraints (8) reduce to the following *R-even inequalities* defined by Ghiani and Laporte [24]:

$$\sum_{e \in \delta(W) \setminus \{e^*\}} x_e \geq x_{e^*} \qquad (W \neq \emptyset,\ W \subset V, |\delta_R(W)| \text{ is even}, e^* \in \delta(W)) \quad (10)$$

These *R-even inequalities* (10) can be explained as follows. Notice first that they are useless when $x_{e^*} = 0$ since we already know from constraints (6) that $\sum_{e \in \delta(W) \setminus \{e^*\}} x_e \geq 0$. So let W be any non-empty proper subset of V such that $|\delta_R(W)|$ is even, and let e^* be any edge in $\delta(W)$ with $x_{e^*} = 1$ (if any). Since $G(x)$ is required to be Eulerian, the number of edges in $G(x)$ having one endpoint in W and the other outside W must be even. These edges that traverse the frontier between W and $V \setminus W$ in $G(x)$ are those in $\delta_R(W)$ as well as the edges $e \in \delta(W)$ with value $x_e = 1$. Since $|\delta_R(W)|$ is supposed to be even and $x_{e^*} = 1$, we can impose $\sum_{e \in \delta(W)} x_e = \sum_{e \in \delta(W) \setminus \{e^*\}} x_e + 1$ to also be even, which means that $\sum_{e \in \delta(W) \setminus \{e^*\}} x_e$ must be greater than or equal to $1 = x_{e^*}$.

Several researchers have implemented cutting plane and Branch & Cut algorithms for the URPP, based on the above formulations. It turns out that cutting planes of type (9) and (10) are easier to generate than the more general ones of type (7) or (8). Ghiani and Laporte have implemented a Branch & Cut algorithm for the URPP, based on connectivity inequalities (2), on R-odd inequalities (9) and on R-even inequalities (10). The separation problem (see Section 2) for connectivity inequalities is solved by means of a heuristic proposed by Fischetti, Salazar and Toth [21]. To separate R-odd inequalities, they use a heuristic inspired by a procedure developed by Grötschel and Win [28]. The exact separation algorithm of [41] could be used to identify violated R-even inequalities, but [24] have developed a faster heuristic procedure that detects several violations at a time. They report very good computational results on a set of 200 instances, corresponding to three classes of random graphs generated as in [31]. Except for 6 instances, the other 194 instances involving up to 350 vertices were solved to optimality in a reasonable amount of time. These results outperform those reported by Christofides, et al. [9], [10] and [36] who solved much smaller randomly generated instances ($|V| \leq 84$).

4. Basic Procedures for the URPP and the UCARP

Up to recently, the best known constructive heuristic for the URPP was due to [22]. This method works along the lines of Christofides's algorithm [8] for the undirected traveling salesman problem, and can be described as follows.

Frederickson's Algorithm

Step 1. Construct a minimum spanning tree S over G^* (see domination relation 3 in Section 3 for the definition of G^*).
Step 2. Determine a minimum cost matching M (with respect to shortest chain costs) on the odd-degree vertices of the graph induced by $R \cup S$.
Step 3. Determine an Eulerian tour in the graph induced by $R \cup S \cup M$.

As Christofides's algorithm for the undirected traveling salesman, the above algorithm has a worst case ratio of 3/2. Indeed, let C^* be the optimal value of the URPP and let C_R, C_S and C_M denote the total cost of the edges in R, S and M, respectively. It is not difficult to show that $C_R + C_S \leq C^*$ and $C_M \leq C^*/2$, and this implies that $C_R + C_S + C_M \leq 3C^*/2$.

Two recent articles [31, 30] contain a description of some basic algorithmic procedures for the design of heuristic methods in an arc routing context. All these procedures are briefly described and illustrated in this section. In what follows, SC_{vw} denotes the shortest chain linking vertex v to vertex w while L_{vw} is the length of this chain. The first procedures, called POSTPONE and REVERSE, modify the order in which edges are serviced or traversed without being serviced.

Procedure POSTPONE

INPUT : a covering tour T with a given orientation and a starting point v on T.
OUTPUT : another covering tour.

Whenever a required edge e appears several times on T, delay service of e until its last occurrence on T, considering v as starting vertex on T.

Procedure REVERSE

INPUT : a covering tour T with a given orientation and a starting point v on T.
OUTPUT : another covering tour.

Step 1. Determine a vertex w on T such that the path linking v to w on T is as long as possible, while the path linking w to v contains all edges in R. Let P denote the path on T from v to w and P' the path from w to v.
Step 2. If P' contains an edge (x,w) entering w which is traversed but not serviced, then the first edges on P' up to (x,w) induce a circuit C. Reverse the orientation of C and go to Step 1.

The next procedure, called SHORTEN, is based on the simple observation that if a tour Tcontains a chain P of traversed (but not serviced) edges, then T can eventually be shortened by replacing P by a shortest chain linking the endpoints of P.

Procedure SHORTEN

INPUT : a covering tour T
OUTPUT : a possibly shorter covering tour.

Step 1. Choose an orientation for T and let v be any vertex on T.
Step 2. Apply POSTPONE and REVERSE
Step 3. Let w be the first vertex on T preceeding a required edge. If L_{vw} is shorter than the length of P, then replace P by SC_{vw}.
Step 4. Repeatedly apply steps 2 and 3, considering the two possible orientations of T, and each possible starting vertex v on T, until no improvement can be obtained.

As an illustration, consider the graph depicted in Figure 2(a) containing 4 required edges shown in bold lines. An oriented tour $T = (c,d,e,f,b,a,e,g,d,c)$ is represented in Figure 2(b) with $v = c$ as starting vertex. Since the required edge (c,d) appears twice on T, POSTPONE makes it first traversed and then serviced, as shown on Figure 2(c). Then, REVERSE determines $P = (c,d,e)$ and $P' = (e,f,b,a,e,g,d,c)$, and since P' contains a non-serviced edge (a,e) entering e, the orientation of the circuit (e,f,b,a,e) is reversed, yielding the new tour represented in Figure 2(d). The first part $P = (c,d,e,a)$ of this tour is shortened into (c,a), yielding the tour depicted in Figure 2(e).

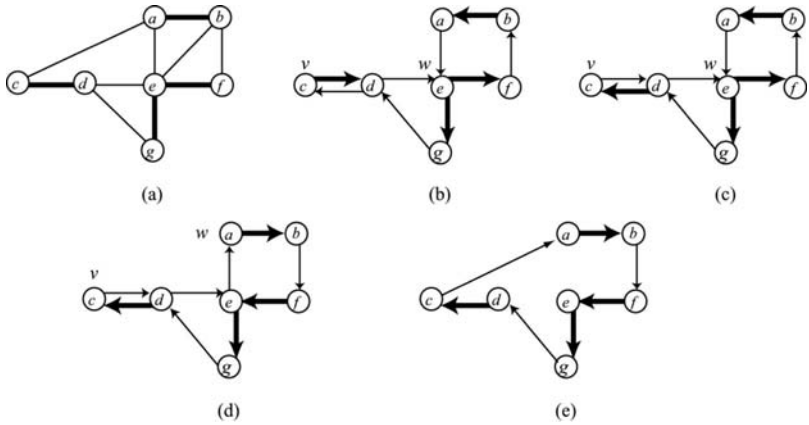


Figure 2. Illustration of procedures POSTPONE, REVERSE and SHORTEN.

The next procedure, called SWITCH, also modifies the order in which required edges are visited on a given tour. It is illustrated in Figure 3.

Procedure SWITCH

INPUT : a covering tour T
OUTPUT : another covering tour.

Step 1. Select a vertex v appearing several times on T.
Step 2. Reverse all minimal cycles starting and ending at v on T.

Given a covering tour T and given a non-required edge (v,w), procedure ADD builds a new tour covering $R \cup (v,w)$. On the contrary, given a required edge (v,w) in R, procedure DROP builds a new tour covering $R \backslash (v,w)$.

Procedure ADD

INPUT : a covering tour T and an edge $(v,w) \notin R$
OUTPUT : a covering tour for $R \cup (v,w)$

Figure 3. Illustration of procedure SWITCH.

Step 1. If neither v nor w appear on T, then determine a vertex z on T minimizing $L_{zv} + L_{wz}$, and add the circuit $SC_{zv} \cup (v,w) \cup SC_{wz}$ on T. Otherwise, if one of v and w (say v), or both of them appear on T, but not consecutively, then add the circuit (v,w,v) on T.
Step 2. Set R: = $R \cup (v,w)$ and attempt to shorten T by means of SHORTEN.

Procedure DROP

INPUT : a covering tour T and an edge (v,w) in R
OUTPUT : a covering tour for $R \backslash (v,w)$.

Step 1. Set R: = R$\backslash(v,w)$.
Step 2. Attempt to shorten T by means of SHORTEN.

The last two procedures, called PASTE and CUT can be used in a UCARP context. PASTE merges two routes into a single tour, possibly infeasible for the UCARP.

Procedure PASTE

INPUT : two routes $T_1 =$ (depot, $v_1, v_2, \ldots, v_r$, depot) and $T_2 =$ (depot, $w_1, w_2, \ldots, w_s$, depot).
OUTPUT : a single route T containing all required edges of R_1 and R_2.

If $(v_r$, depot) and (depot, $w_1)$ are non-serviced edges on T_1 and T_2, respectively, then set $T =$ (depot, $v_1, \ldots, v_r, w_1, \ldots, w_s$, depot), else set $T =$ (depot, $v_1, \ldots, v_r$, depot, $w_1, \ldots, w_s$, depot).

CUT decomposes a non-feasible route into a set of feasible routes (i.e., the total demand on each route does not exceed the capacity Q of each vehicle).

Procedure CUT

INPUT : A route T starting and ending at the depot, and covering R.
OUTPUT : a set of feasible vehicle routes covering R.

Step 0. Label the vertices on T so that $T =$ (depot, $v_1, v_2, \ldots, v_t$, depot).
Step 1. Let D denote the total demand on T. If $D \leq Q$ then STOP : T is a feasible vehicle route.
Step 2. Determine the largest index s such that (v_{s-1}, v_s) is a serviced edge, and the total demand on the path (depot, $v_1, \ldots, v_s$) from the depot to v_s does not exceed Q. Determine the smallest index r such that (v_{r-1}, v_r) is a serviced edge and the total demand on the path $(v_r, \ldots, v_t$, depot) from v_r to the depot does not exceed $Q(\lceil D/Q \rceil - 1)$. If $r > s$ then set $r = s$.

For each index q such that $r \leq q \leq s$, let v_q* denote the first endpoint of a required edge after v_q on T, and let δ_q denote the length of the chain linking v_q to v_q* on T. Select the vertex v_q minimizing $L(v_q) = L_{v_q,\,\text{depot}} + L_{\text{depot},\,v_q*} - \delta_q$.

Step 3. Let P_{v_q} (P_{v_q*}) denote the paths on T from the depot to v_q (v_q*). Construct the feasible vehicle route made of $P_{v_q} \cup SC_{v_q,\text{depot}}$, replace P_{v_q*} by SC_{depot,v_q*} on T and return to Step 1.

The main idea of the above algorithm is to try to decompose the non-feasible route T into $\lceil D/Q \rceil$ feasible vehicle routes, where $\lceil D/Q \rceil$ is a trivial lower bound on the number of vehicles needed to satisfy the demand on T. If such a decomposition exists, then the demand covered by the first vehicle must be large enough so that the residual demand for the $\lceil D/Q \rceil - 1$ other vehicles does not exceed $Q(\lceil D/Q \rceil - 1)$ units: this constraint defines the above vertex v_r. The first vehicle can however not service more than Q units, and this defines the above vertex v_s. If $r > s$, this means that it is not possible to satisfy the demand with $\lceil D/Q \rceil$ vehicle routes, and the strategy described above is to cover as many required edges as possible with the first vehicle. Otherwise, the first vehicle satisfies the demand up to a vertex v_q on the path linking v_r to v_s, and the process is then repeated on the tour T'obtained from T by replacing the path (depot, $v_1, \ldots, v_q$*) by a shortest path from the depot to v_q*. The choice for v_q is made so that the length of T' plus the length of the first vehicle route is minimized.

Procedure CUT is illustrated in Figure 4. The numbers in square boxes are demands on required edges. The numbers on the dashed lines or on the edges are shortest chain lengths. In this example, $Q = 11$ and $D = 24$. The procedure first computes $Q(\lceil D/Q \rceil - 1) = 22$, which implies that the first vehicle route must include at least the first required edge (i.e., $r = 2$). Since the first vehicle cannot include more than the three first required edges without having a weight exceeding Q, we have $s = 5$. Now, $v_2* = v_3$, $v_3* = v_3$, $v_4* = v_4$ and $v_5* = v_6$, and since $L(v_2) = 10$, $L(v_3) = 12$, $L(v_4) = 8$ and $L(v_5) = 11$, vertex v_4 is selected. The first vehicle route is therefore equal to (depot,v_1,v_2,v_3,v_4,depot) and the procedure is reapplied on the tour (depot, $v_4, v_5, \ldots, v_{10}$, depot) with a total demand $D = 17$. We now have $s = 7$ and $r = 9$, which means that the four remaining required edges cannot be serviced by two vehicles. We therefore set $s = r = 7$, which means that the second vehicle route is equal to (depot, v_4, v_5, v_6, v_7, depot) and the procedure is repeated on $T =$(depot, v_8, v_9, v_{10}, depot) with $D = 12$. Since $s = r = 9$, the third vehicle route is equal to (depot, v_8, v_9, depot) and the residual tour $T =$(depot, v_9, v_{10}, depot) is now feasible and corresponds to the fourth vehicle route.

Notice that procedure CUT does not necessarily produce a solution with a minimum number of vehicle routes. Indeed, in the above example, the initial route

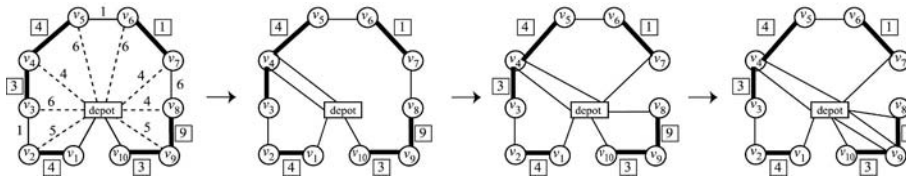


Figure 4. Illustration of procedure CUT.

T has been decomposed into four vehicle routes while there exists a solution with three vehicle routes (depot, $v_1, \ldots, v_5$, depot), (depot, $v_6, \ldots, v_9$, depot) and (depot, v_9, v_{10}, depot).

5. Recent Heuristic Algorithms for the URPP and the UCARP

The procedures described in the previous section can be used as basic tools for the design of constructive algorithms for the URPP. As an example, a solution to the URPP can easily be obtained by means of the following very simple algorithm designed by [31].

Algorithm Construct-URPP

Step 1. Choose a required edge (v_i,v_j) and set $T = (v_i,v_j,v_i)$.
Step 2. If T contains all required edges then stop; else chose a required edge which is not yet in T and add it to T by means of procedure ADD.

Post-optimization procedures can be designed on the basis of procedures DROP, ADD and SHORTEN. As an example, an algorithm similar to the 2-opt procedure [11] for the undirected traveling salesman problem can be designed for the URPP as shown below.

Algorithm 2-opt-URPP

Step 1. Choose an orientation of the given tour T and select two arcs (v_i,v_j) and (v_r,v_s) on T. Build a new tour T' by replacing these two arcs by the shortest chains SP_{ir} and SP_{js}, and by reversing the orientation of the path linking v_j to v_r on T.
Step 2. Let R' be the set of required edges appearing on T'. Apply SHORTEN to determine a possibly shorter tour T'' that also covers R'. If $R \neq R'$ then add the missing required edges on T'' by means of procedure ADD.
Step 3. If the resulting tour T'' has a lower cost than T, then set T equal to T''.
Step 4. Repeat steps 1, 2 and 3 with the two possible orientations of T and with all possible choices for (v_i,v_j) and (v_r,v_s), until no additional improvement can be obtained.

[31] propose to use a post-optimization procedure, called DROP-ADD, similar to the Unstringing-Stringing (US) algorithm for the undirected traveling salesman problem [23]. DROP-ADD tries to improve a given tour by removing a required edge and reinserting it by means of DROP and ADD, respectively.

Algorithm DROP-ADD

Step 1. Choose a required edge e, and build a tour T' covering $R\backslash\{e\}$ by means of DROP.

Step 2. If edge e is not traversed on T', then add e to T' by means of ADD.
Step 3. If the resulting tour T' has a lower cost than T, then set T equal to T'.
Step 4. Repeat steps 1, 2 and 3 with all possible choices for e, until no additional improvement can be obtained.

[31] have generated 92 URPP instances to test the performance of these two post-optimization procedures. These 92 instances correspond to three classes of randomly generated graphs. First class graphs are obtained by randomly generating points in the plane; class 2 graphs are grid graphs generated to represent the topography of cities, while class 3 contains grid graphs with vertex degrees equal to 4. Computational experiments show that Frederickson's algorithm is always very quick but rarely optimal. Percentage gaps with respect to best known solutions can be as large as 10%, particularly in the case of larger instances or when the number of connected components in G_R is large. Applying DROP-ADD after Frederickson's algorithm typically generates a significant improvement within a very short computing time. However, much better results are obtained if 2-opt-URPP is used instead of DROP-ADD, but computing times are then more significant. The combination of Frederickson's algorithm with 2-opt-URPP has produced 92 solutions which are now proved to be optimal using the Branch & Cut algorithm of [24].

Local search techniques are iterative procedures that aim to find a solution s minimizing an objective function f over a set S of feasible solutions. The iterative process starts from an initial solution in S, and given any solution s, the next solution is chosen in the *neighbourhood* $N(s) \subseteq S$. Typically, a neighbour s' in $N(s)$ is obtained from s by performing a local change on it. *Simulated Annealing* [34] and *Tabu Search* [25] are famous local search techniques that appear to be quite successful when applied to a broad range of practical problem.

[30] have designed an adaptation of Tabu Search, called CARPET, for the solution of the UCARP. Tests on benchmark problems have shown that CARPET is a highly efficient heuristic. The algorithm works with two objective functions: $f(s)$, the total travel cost, and a penalized objective function $f'(s) = f(s) + \alpha E(s)$, where α is a positive parameter and $E(s)$ is the total excess weight of all routes in a possibly infeasible solution s. CARPET performs a search over neighbor solutions, by moving at each iteration from the current solution to its best non-tabu neighbor, even if this causes a deterioration in the objective function. A neighbor solution is obtained by moving a required edge from its current route to another one, using procedures DROP and ADD.

Recently, [39] have designed a new local search technique called *Variable Neighborhood Search* (VNS). The basic idea of VNS is to consider several neighborhoods for exploring the solution space, thus reducing the risk of becoming trapped in a local optimum. Several variants of VNS are described in [29]. We describe here the simplest one which performs several descents with different neighborhoods until a local optimum for all considered neighborhoods is reached. This particular variant of VNS is called *Variable neighborhood descent* (VND). Let $N_1, N_2, \ldots, N_K$ denote

a set of K neighborhood structures (i.e., $N_i(s)$ contains the solution that can be obtained by performing a local change on s according to the i-th type). VND works as follows.

Variable Neighbourhood Descent

Step 1. Choose an initial solution s in S.
Step 2. Set i: = 1 and s_{best}: = s.
Step 3. Perform a descent from s, using neighborhood N_i, and let s' be the resulting solution. If $f(s') < f(s)$ then set s: = s'. Set i: = $i + 1$. If $i \leq K$ then repeat Step 3.
Step 4. If $f(s) < f(s_{best})$ then go to Step 2; else stop.

[32] have designed an adaptation of VND to the undirected CARP, called VND-CARP. The search space S contains all solutions made of a set of vehicle routes covering all required edges and satisfying the vehicle capacity constraints. The objective function to be minimized on S is the total travel cost. The first neighborhood $N_1(s)$ contains solutions obtained from s by moving a required edge (v,w) from its current route T_1 to another one T_2. Route T_2 either contains only the depot (i.e., a new route is created), or a required edge with an endpoint distant from v or w by at most α, where α is the average length of an edge in the network. The addition of (v,w) into T_2 is performed only if there is sufficient residual capacity on T_2 to integrate (v,w). The insertion of (v,w) into T_2 and the removal of (v,w) from T_1 are performed using procedures ADD and DROP described in the previous section.

A neighbor in $N_i(s)$ ($i > 1$) is obtained by modifying i routes in s as follows. First, a set of i routes in s are merged into a single tour using procedure PASTE, and procedure SWITCH is applied on it to modify the order in which the required edges are visited. Then, procedure CUT divides this tour into feasible routes which are possibly shortened by means of SHORTEN.

As an illustration, consider the solution depicted in Figure 5(a) with three routes T_1 = (depot,a,b,c,d,depot), T_2 = (depot,b,e,f,b,depot) and T_3 = (depot,g,*h,depot)*. The capacity Q of the vehicles is equal to 2, and each required edge has a unit demand. Routes T_1 and T_2 are first merged into a tour T = (depot,a,b,c,d, depot,b,e,f,b,depot) shown in Figure 5(b). Then, SWITCH modifies T into T' = (depot,d,c,b,f,e,b,a,depot,b,depot) represented in Figure 5(c). Procedure CUT divides T' into two feasible routes T'_1 = (depot,d,c,b,f,e,b,depot) and T'_2 = (depot,b,a,depot,b,depot) depicted in Figure 5(d). Finally, these two routes are shortened into T''_1 = (depot,d,c,f,e,b,depot) and T''_2 = (depot,b,a,depot) using SHORTEN. Routes T''_1 and T''_2 together with the third non-modified route T_3 in s constitute a neighbor of s in $N_2(s)$ shown in Figure 5(e).

Hertz and Mittaz have performed a comparison between CARPET, VND-CARP and the following well known heuristics for the UCARP : CONSTRUCT-STRIKE

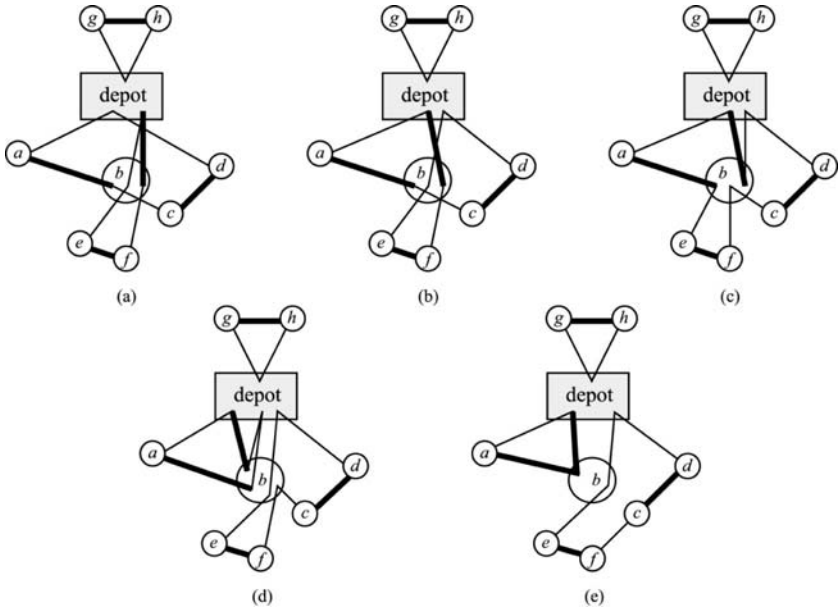


Figure 5. Illustration of neighbourhood N_2.

[7], PATH-SCANNING [26], AUGMENT-MERGE [27], MODIFIED-CONSTRUCT-STRIKE [43] and MODIFIED-PATH-SCANNING [43]. Three sets of test problems have been considered. The first set contains 23 problems described in [13] with $7 \leq |V| \leq 27$ and $11 \leq |E| \leq 55$, all edges requiring a service (i.e. $R = E$). The second set contains 34 instances supplied by Benavent [5] with $24 \leq |V| \leq 50$, $34 \leq |E| \leq 97$ and $R = E$. The third set of instances was generated by Hertz, Laporte and Mittaz [30] in order to evaluate the performance of CARPET. It contains 270 larger instances having 20, 40 or 60 vertices with edge densities in [0.1,0.3], [0.4,0.6] or [0.7,0.9] and $|R|/|E|$ in [0.1,0.3], [0.4,0.6] or [0.8,1.0]. The largest instance contains 1562 required edges.

A lower bound on the optimal value was computed for each instance. This lower bound is the maximum of the three lower bounds CPA, LB2′ and NDLB′ described in the literature. The first, CPA, was proposed by Belenguer and Benavent [4] and is based on a cutting plane procedure. The second and third, LB2′ and NDLB′, are modified versions of LB2 [6] and NDLB [33] respectively. In LB2′ and NDLB′, a lower bound on the number of vehicles required to serve a subset R of edges is computed by means of the lower bounding procedure LR proposed by Martello and Toth [37] for the bin packing problem, instead of $\lceil D/Q \rceil$ (where D is the total demand on R).

Average results are reported in tables 1 and 2 with the following information:

- *Average deviation*: average ratio (in %) of the heuristic solution value over the best known solution value.

Table 1. Computational results on DeArmon instances

	Ps	Am	Cs	Mcs	Mps	Carpet	Vnd
Average deviation	7.26	5.71	14.03	4.02	4.45	0.17	0.17
Worst deviation	22.27	25.11	43.01	40.83	23.58	2.59	1.94
Number of proven optima	2	3	2	11	5	18	18

- *Worst deviation*: largest ratio (in %) of the heuristic solution value over the best known solution value;
- *Number of proven optima*: number of times the heuristic has produced a solution value equal to the lower bound.

Ps, Am, Cs, MCs, Mps and Vnd are abbreviations for Path-Scanning, Augment-Merge, Construct-Strike, Modified-Construct-Strike, Modified-Path-Scanning and Vnd-Carp.

It clearly appears in Table 1 that the tested heuristics can be divided into three groups. Construct-Strike, Path-Scanning and Augment-Merge are constructive algorithms that are not very robust: their average deviation from the best known solution value is larger than 5%, and their worst deviation is larger than 20%. The second group contains Modified-Construct-Strike and Modified-Path-Scanning; while better average deviations can be observed, the worst deviation from the best known solution value is still larger than 20%. The third group contains algorithms Carpet and Vnd-Carp that are able to generate proven optima for 18 out of 23 instances.

It can be observed in Table 2 that Vnd-Carp is slightly better than Carpet both in quality and in computing time. Notice that Vnd-Carp has found 220 proven optima out of 324 instances. As a conclusion to these experiments, it can be observed that the most powerful heuristic methods for the solution of the UCARP all employ on the basic tools described in Section 4.

6. Conclusion and Future Developments

In the field of exact methods, Branch & Cut has known a formidable growth and considerable success on many combinatorial problems. Recent advances made by

Table 2. Computational results on Benavent and Hertz-Laporte-Mittaz instances

	Benavent instances		Hertz-Laporte-Mittaz instances	
	Carpet	Vnd	Carpet	Vnd
Average deviation	0.93	0.54	0.71	0.54
Worst deviation	5.14	2.89	8.89	9.16
Number of proven optima	17	17	158	185
Computing times in seconds	34	21	350	42

Corberán and Sanchis [10], Letchford [36] and Ghiani and Laporte [24] indicate that this method also holds much potential for arc routing problems.

In the area of heuristics, basic simple procedures such as POSTPONE, REVERSE, SHORTEN, DROP, ADD, SWITCH, PASTE and CUT have been designed for the URPP and UCARP [31]. These tools can easily be adapted to the directed case [38]. Powerful local search methods have been developed for the UCARP, one being a Tabu Search [30], and the other one a Variable Neighborhood Descent [32].

Future developments will consist in designing similar heuristic and Branch & Cut algorithms for the solution of more realistic arc routing problems, including those defined on directed and mixed graphs, as well as problems incorporating a wider variety of practical constraints.

References

[1] A.A. Assad and B.L. Golden, Arc Routing Methods and Applications, in M.O. Ball, T.L. Magnanti, C.L. Monma and G.L. Nemhauser (eds.), *Network Routing,* Handbooks in Operations Research and Management Science, North-Holland, Amsterdam, (1995).

[2] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam, (1973).

[3] F. Barahona and M. Grötschel, On the Cycle Polytope of a Binary Matroid, *Journal of Combinatorial Theory* 40: 40–62 (1986).

[4] J.M. Belenguer and E. Benavent, A Cutting Plane Algorithm for the Capacitated Arc Routing Problem, Unpublished manuscript, University of Valencia, Spain, (1997).

[5] E. Benavent, ftp://indurain.estadi.uv.ed/pub/CARP, (1997).

[6] E. Benavent, V. Campos, A. Corberán and E. Mota, The Capacitated Arc Routing Problem: Lower Bounds, *Networks* 22: 669–690 (1992).

[7] N. Christofides, The Optimum Traversal of a Graph, *Omega* 1: 719–732 (1973).

[8] N. Christofides, Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem, Report No 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, (1976).

[9] N. Christofides, V. Campos, A. Corberán and E. Mota, An Algorithm for the Rural Postman Problem, Imperial College Report I C.O.R. 81.5, London, (1981).

[10] A. Corberán and J.M. Sanchis, A Polyhedral Approach to the Rural Postman Problem, *European Journal of Operational Research* 79: 95–114 (1994).

[11] G.A. Croes, A Method for Solving Traveling-Salesman Problems, *Operations Research* 6: 791–812 (1958).

[12] G.B. Dantzig, D.R. Fulkerson and S.M. Johnson, Solution of a Large Scale Traveling Salesman Problem, *Operations Research* 2: 393–410 (1954).

[13] J.S. DeArmon, A Comparison of Heuristics for the Capacitated Chinese Postman Problem, Master's Thesis, University of Maryland at College Park, (1981).

[14] M. Dror, *ARC Routing : Theory, Solutions and Applications*, Kluwer Academic Publishers, Boston, (2000).

[15] M. Dror, H. Stern and P. Trudeau Postman Tour on a Graph with Precedence Relation on Arcs, *Networks* 17: 283–294 (1987).

[16] J. Edmonds and E.L. Johnson, Matching, Euler Tours and the Chinese Postman Problem, *Mathematical Programming* 5: 88–124 (1973).

[17] R.W. Eglese, Routing Winter Gritting Vehicles, *Discrete Applied Mathematics* 48: 231–244 (1994).

[18] R.W. Eglese and L.Y.O. Li, A Tabu Search Based Heuristic for Arc Routing with a Capacity Constraint and Time Deadline, in I.H. Osman and J.P. Kelly (eds.), *Meta-Heuristics: Theory & Applications,* Kluwer Academic Publishers, Boston, (1996).

[19] H.A. Eiselt, M. Gendreau and G. Laporte, Arc Routing Problems, Part 1: The Chinese Postman Problem, *Operations Research* 43: 231–242 (1995a).

[20] H.A. Eiselt, M. Gendreau and G. Laporte, Arc Routing Problems, Part 2: The Rural Postman Problem, *Operations Research* 43: 399–414 (1995b).

[21] M. Fischetti, J.J. Salazar and P. Toth, A Branch-and Cut Algorithm for the Symmetric Generalized Traveling Salesman Problem, *Operations Research* 45: 378–394 (1997).

[22] G.N. Frederickson, Approximation Algorithms for Some Postman Problems, *Journal of the A.C.M.* 26: 538–554 (1979).

[23] M. Gendreau, A. Hertz and G. Laporte, A Tabu Search Heuristic for the Vehicle Routing Problem, *Management Science* 40: 1276–1290 (1994).

[24] G. Ghiani and G. Laporte, A Branch-and-cut Algorithm for the Undirected Rural Postman Problem, *Mathematical Programming* 87: 467–481 (2000).

[25] F. Glover, Tabu Search – Part I, *ORSA Journal on Computing* 1: 190–206 (1989).

[26] B.L. Golden, J.S. DeArmon and E.K. Baker, Computational Experiments with Algorithms for a Class of Routing Problems, *Computers & Operations Research* 10: 47–59 (1983).

[27] B.L. Golden and R.T. Wong, Capacitated Arc Routing Problems, *Networks* 11: 305–315 (1981).

[28] M. Grötschel and Z. Win, A Cutting Plane Algorithm for the Windy Postman Problem, *Mathematical Programming* 55: 339–358 (1992).

[29] P. Hansen and N. Mladenović, An Introduction to VNS. In S. Voss, S. Martello, I.H. Osman and C. Roucairol (eds.), *Meta-Heuristics : Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston, (1998).

[30] A. Hertz, G. Laporte and M. Mittaz, A Tabu Search Heuristic for the Capacitated Arc Routing Problem, *Operations Research* 48: 129–135 (2000).

[31] A. Hertz, G. Laporte and P. Nanchen-Hugo, Improvement Procedure for the Undirected Rural Postman Problem, *INFORMS Journal on Computing* 11: 53–62 (1999).

[32] A. Hertz and M. Mittaz, A Variable Neighborhood Descent Algorithm for the Undirected Capacitated Arc Routing Problem, *Transportation Science* 35: 425–434 (2001).

[33] R. Hirabayashi, Y. Saruwatari and N. Nishida, Tour Construction Algorithm for the Capacitated Arc Routing Problem, *Asia-Pacific Journal of Operational Research* 9: 155–175 (1992).

[34] S. Kirkpatrick, C.D. Gelatt and M. Vecchi, M. Optimization by Simulated Annealing, *Science* 220: 671–680 (1983).

[35] J.K. Lenstra and A.H.G. Rinnooy Kan, On General Routing Problems, *Networks* 6: 273–280 (1976).

[36] A.N. Letchford, Polyhedral Results for some Constrained Arc Routing Problems, PhD Dissertation, Dept. of Management Science, Lancaster University, (1997).

[37] S. Martello and P. Toth, Lower Bounds and Reduction Procedures for the Bin Packing Problem, *Discrete Applied Mathematics* 28: 59–70 (1990).

[38] M. Mittaz, Problèmes de Cheminements Optimaux dans des Réseaux avec Contraintes Associées aux Arcs, PhD Dissertation, Department of Mathematics, Ecole Polytechnique Fédérale de Lausanne, (1999).

[39] N. Mladenović and P. Hansen, Variable Neigbhourhood Search, *Computers & Operations Research* 34: 1097–1100 (1997).

[40] C.S. Orloff, A Fundamental Problem in Vehicle Rouring, *Networks* 4: 35–64 (1974).

[41] M.W. Padberg and M.R. Rao, Odd Minimum Cut-Setss and *b*-Matchings, *Mathematics for Operations Research* 7: 67–80 (1982).

[42] M.W. Padberg and G. Rinaldi, A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems, *SIAM Review* 33: 60–100 (1991).

[43] W.-L. Pearn, Approximate Solutions for the Capacitated Arc Routing Problem, *Computers & Operations Research* 16: 589–600 (1989).

[44] S. Roy and J.M. Rousseau, The Capacitated Canadian Postman Problem, *INFOR* 27: 58–73 (1989).

[45] L.A. Wolsey, *Integer Programming*, John Wiley & Sons, (1998).

10

Software and Hardware Testing Using Combinatorial Covering Suites

Alan Hartman
IBM Haifa Research Laboratory

Abstract

In the 21st century our society is becoming more and more dependent on software systems. The safety of these systems and the quality of our lives is increasingly dependent on the quality of such systems. A key element in the manufacture and quality assurance process in software engineering is the testing of software and hardware systems. The construction of efficient combinatorial covering suites has important applications in the testing of hardware and software. In this paper we define the general problem, discuss the lower bounds on the size of covering suites, and give a series of constructions that achieve these bounds asymptotically. These constructions include the use of finite field theory, extremal set theory, group theory, coding theory, combinatorial recursive techniques, and other areas of computer science and mathematics. The study of these combinatorial covering suites is a fascinating example of the interplay between pure mathematics and the applied problems generated by software and hardware engineers. The wide range of mathematical techniques used, and the often unexpected applications of combinatorial covering suites make for a rewarding study.

1. Introduction

Testing is an important but expensive part of the software and hardware development process. In order to test a large software or hardware system thoroughly, many sequences of possible inputs must be tried, and then the expected behavior of the system must be verified against the system's requirements. This is usually a labor-intensive process that requires a great deal of time and resources. It has often been estimated that testing consumes at least 50% of the cost of developing a new piece of software. The testing costs for hardware and safety-critical systems are often higher.

The consequences of inadequate testing can be catastrophic. An extreme example is the software failure in the Therac-5 radiation therapy machine [27] that is known to have caused six massive overdoses of radiation to be administered to cancer patients resulting in deaths and severe injuries. A further example of a catastrophic software failure

occurred when the Ariane 5 satellite launcher exploded 40 seconds into its maiden flight. A register overflow failure which occurred simultaneously on one processor and on its backup [29] caused both processors to shut down, and eventually abort the satellite mission. The most celebrated hardware bug is the Pentium floating point division bug [14] which caused an error in the accuracy of a small number of division computations. This in itself, does not sound like a disaster – but the cost to the Intel corporation was measured in millions of dollars.

An approach to lowering the cost of software testing was put forward by Cohen, Dalal, Fredman, and Patton [11] using test suites generated from combinatorial designs. This approach involves identifying parameters that define the space of possible test scenarios, then selecting test scenarios in such a way as to cover all the pairwise (or t-wise) interactions between these parameters and their values. A similar approach was used earlier in hardware testing by Tang, Chen, and Woo [41,42] and Boroday and Grunskii [3]. The approach is familiar to statisticians, and has been used in the design of agricultural experiments since the 1940s [17]. The statistical analysis of such experiments is facilitated if every interaction is covered precisely the same number of times, however Cohen et al. point out that in software testing it is often sufficient to generate test suites so that each interaction is covered *at least* once rather than insisting on the more restrictive condition required by the statisticians.

As an example, consider the testing of an internet site that must function correctly on three operating systems (Windows, Linux, and Solaris), two browsers (Explorer and Netscape), three printers (Epson, HP, and IBM), and two communication protocols (Token Ring and Ethernet). Although there are $36 = 3 \times 2 \times 3 \times 2$ possible test configurations, the nine tests in Figure 1 cover all the pairwise interactions between different parameters of the system.

The interactions between operating systems and printers are all covered precisely once, but some interactions between operating systems and browsers are covered more than once. For example, Windows and Explorer are tested together twice in the test suite.

More generally, if a software system has k parameters, each of which must be tested with n_i values ($1 \leq i \leq k$), then the total number of possible test vectors is the product $\prod_i n_i$. If we wish to test the interactions of any subset of t parameters, then the number of test vectors may be as small as the product of the t largest values n_i.

The same argument applies to testing software that computes a function with k parameters, or a piece of hardware with k input ports. In the context of hardware testing it is of particular importance to find small sets of binary vectors of length k with the property that any fixed set of t coordinate places contains all 2^t binary strings of length t. In Figure 2 we illustrate a set of 8 binary vectors of length 4 such that any 3 coordinate places contain all possible binary strings of length 3.

Operating System	Browser	Printer	Protocol
Windows	Explorer	Epson	Token Ring
Windows	Netscape	HP	Ethernet
Windows	Explorer	IBM	Ethernet
Linux	Netscape	Epson	Token Ring
Linux	Explorer	HP	Ethernet
Linux	Netscape	IBM	Token Ring
Solaris	Explorer	Epson	Ethernet
Solaris	Netscape	HP	Token Ring
Solaris	Explorer	IBM	Ethernet

Figure 1. A set of test cases with pairwise coverage.

In the next section we will formalize the problem of finding minimal covering suites. We then discuss various techniques for constructing good covering suites using finite fields (in Section 3), extremal set theory (in Section 4), group theory (Section 6), coding theory (Section 4), algorithmic methods (Sections 5 and 8), and combinatorial recursive methods (Section 7). We also discuss briefly the results on lower bounds on the sizes of covering suites in Section 4. Finally in Section 9 we close with an account of three diverse applications of covering suites including one in an area far removed from the original motivating problem.

The wide range of methods used, and the variety of applications of these combinatorial objects provide evidence of the value of interdisciplinary studies, and the cross-fertilization that occurs between mathematics and computer science.

0000
0011
0101
0110
1001
1010
1100
1111

Figure 2. A covering suite of strength 3 with four binary parameters.

2. Covering Suites and Their Properties

Let $D_1, D_2, \ldots, D_k$ be finite sets of cardinalities $n_1, n_2, \ldots, n_k$ respectively. A *test suite* with N test vectors is an array $A = (a_{ij} : 1 \leq i \leq N, 1 \leq j \leq k)$ where each member of the array $a_{ij} \in D_j$ for all i and j. The rows of the array are called *test vectors* or *test cases* or just simply *tests*. The set D_i is the domain of possible values for the i-th coordinate of the test vector.

We shall say that the test suite A is a *t-wise covering suite* with parameters $n_1, n_2, \ldots, n_k$ if for any t distinct columns $c_1, c_2, \ldots, c_t$ and for any ordered t-tuple $T \in D_{c_1} \times D_{c_2} \times \ldots \times D_{c_t}$ there exists at least one row r such that $(a_{rc_1}, a_{rc_2}, \ldots, a_{rc_t}) = T$.

We define the *covering suite number* $CS_t(n_1, n_2, \ldots, n_k)$ to be the minimum integer N such that there exists a t-wise covering suite with N test cases for k domains of sizes $n_1, n_2, \ldots, n_k$. The function is well-defined, since the actual members of the sets D_j are not important; what really matters is the cardinalities of the sets. Unless otherwise stated, we will assume that $D_j = \{0, 1, \ldots, n_j - 1\}$.

If all the domains are the same size, say n, we will denote $CS_t(n, n, \ldots, n)$ by $CS_t(n^k)$ and we also use this standard exponential notation for multi-sets in other contexts, so that for example, we will use $CS_t(n^2, m^3)$ for $CS_t(n, n, m, m, m)$.

A strict interpretation of the definition implies that $CS_0(n^k) = 1$, since at least one row is required to cover the empty 0-tuple. It is also straightforward to see that $CS_1(n^k) = n$, since each column of the minimal array must contain a permutation of I_n.

In the rest of this section, we will establish some elementary properties of covering suites and the covering suite numbers defined above.

Lemma 2.1 $CS_t(n_1, n_2, \ldots, n_k) \geq n_1 n_2 \ldots n_t$, and hence $n^k \geq CS_t(n^k) \geq n^t$

Proof. Consider the number of test cases required to cover all the combinations of values in the first t domains. Details are left as an exercise. □

We now show that $CS_t(n^k)$ is a non-decreasing function of t, n, and k.

Lemma 2.2 For all positive integer parameters, we have:

a) if $k < r$ then $CS_t(n^k) \leq CS_t(n^r)$
b) if $n_i \leq m_i$ for all i then $CS_t(n_1, n_2, \ldots n_k) \leq CS_t(m_1, m_2, \ldots m_k)$
c) if $n < m$ then $CS_t(n^k) < CS_t(m^k)$
d) if $s < t$ then $CS_s(n^k) \leq CS_t(n^k)$.

Proof. a) Let $CS_t(n^r) = N$ and let A be a t-wise covering suite with N test cases for r domains of size n. Deleting $r - k$ columns from A leaves a t-wise covering test suite with N test cases for k domains of size n, and thus $CS_t(n^k) \leq N = CS_t(n^r)$.

b) Let $CS_t(m_1, m_2, \ldots, m_k) = N$ and let A be a t-wise covering suite with N test cases for the k domains of size m_i. Replace every entry of A that lies in the set $I_{m_i} - I_{n_i}$ by an arbitrary member of I_{n_i} to produce a t-wise covering suite with N test cases for the k domains of size n_i, thus $CS_t(n_1, n_2, \ldots, n_k) \leq N = CS_t(m_1, m_2, \ldots, m_k)$.

c) To prove the strict inequality in c), we observe that the symbols in any column may be permuted independently of each other without affecting the coverage property. We permute the symbols so that the first row of the larger array is the constant vector with value $m - 1$ (the largest member of I_m). Now delete this row, and proceed as in the proof of part b).

d) This follows from the fact that every s-tuple is contained in some t-tuple. Moreover, the inequality is strict when $n > 1$ and $k \geq t$. □

The following result shows that there is a stronger relationship between the sizes of covering suites when increasing their strength t.

Lemma 2.3 We have $CS_t(n_1, n_2, \ldots, n_k) \geq n_1 CS_{t-1}(n_2, n_3, \ldots, n_k)$ and thus $CS_t(n^k) \geq nCS_{t-1}(n^{k-1})$.

Proof. Consider the n_1 sub-arrays of a t-wise covering array consisting of all rows where the first column takes a constant value, and delete the first column, see Figure 3. Each such sub-array must be a $(t-1)$-wise covering array, which implies the result. □

The problem of minimizing the number N of test cases in a t-wise covering test suite for k domains of size n was apparently first studied by Renyi [35], and many papers on the subject have appeared since then. Many of these consider the mathematically equivalent problem of maximizing the number k of domains of size n in a t-wise covering test suite with a fixed number N of test cases. This is known as the problem of finding the size of a largest *family of t-independent n-partitions of an N-set*. Other names used in the literature for test suites are *covering arrays, (k,t)-universal sets*, and *t-surjective arrays*.

3. Orthogonal Arrays and Finite Fields

Orthogonal arrays are structures that have been used in the design of experiments for over 50 years. An orthogonal array of size N with k constraints, n levels, strength t, and index λ is an $N \times k$ array with entries from $I_n = \{0, 1, \ldots n-1\}$ with the property that: in every $N \times t$ submatrix, every $1 \times t$ row vector appears precisely $\lambda = N/n^t$ times. See Figure 4.

In a fundamental paper, Bush [6] gave constructions for orthogonal arrays of index 1, and bounds on their parameters. It is clear that an orthogonal array of index 1 is a special case of a covering suite, since in a covering suite each $1 \times t$ row vector

0 0 0	$\geq CS_{t-1}(n_2, n_3, ..., n_k)$
1 1 1	$\geq CS_{t-1}(n_2, n_3, ..., n_k)$
. . .	$\geq CS_{t-1}(n_2, n_3, ..., n_k)$
$n_1 - 1$ $n_1 - 1$	$\geq CS_{t-1}(n_2, n_3, ..., n_k)$

Figure 3. Proof of Lemma 2.3.

is required to appear *at least* once. Thus, an orthogonal array is always a minimal covering suite.

Orthogonal arrays of strength 2 and index 1 have been especially well studied as they are equivalent to mutually orthogonal Latin squares of order n.

A *Latin square* of *order n* is a square array of side n with entries from the set I_n with the property that every row and every column contains every member of the set precisely once. Two Latin squares of order n are said to be mutually orthogonal if for any ordered pair of elements $(x, y\} \in I_n^2$ there exists precisely one cell, such that the first square has the value x in the cell, and the second square has the value y in that cell. We illustrate two mutually orthogonal Latin squares of order 3 in Figure 5 below. Notice that all nine pairs of symbols (x, y) occur once in the same cell of the squares.

0000
0111
0222
1021
1102
1210
2012
2120
2201

Figure 4. An orthogonal array of strength 2.

0 1 2	**0 1 2**
2 0 1	**1 2 0**
1 2 0	**2 0 1**

Figure 5. A pair of mutually orthogonal Latin squares of side 3.

A set of $k-2$ Latin squares of order n, each of which is orthogonal to the other, can be put in one-to-one correspondence with an orthogonal array of size n^2 with k constraints, n levels, strength 2, and index 1, as follows.

We define a row of the array for each of the n^2 cells in the Latin squares. The first column of each row contains the row number of the cell, the second column contains the column number of the cell, and the j-th column (for $j > 2$) contains the element in the cell of the j-2nd Latin square. We illustrate this construction using the two orthogonal Latin squares of order 3 given above to build the orthogonal array of size 9, with 4 constraints, 3 levels, strength 2, and index 1, given at the beginning of this section (Figure 4).

It is well-known (see, for example, [12]) that there exists a set of $n-1$ mutually orthogonal Latin squares of order n if and only if there exists a finite projective plane of order n, and that, moreover, the number of mutually orthogonal Latin squares of order n is at most $n-1$. We will see below that there are many values of n for which this holds, and they will help us to construct covering suites in many cases. We summarize this in the following result:

Theorem 3.1 $CS_2(n^k) = n^2$ for all $k \leq n+1$ if and only if there exists a projective plane of order n, and $CS_2(n^k) > n^2$ for all $k > n+1$.

Proof. If there exists a projective plane of order n, then there exist $n-1$ mutually orthogonal Latin squares of order n, which implies the existence of an orthogonal array of strength 2, index 1, $n+1$ constraints, and n levels. Hence $CS_2(n^k) \leq n^2$ for all $k \leq n+1$, using the monotonicity results (Lemma 2.2). But, by Lemma 2.1, we have $CS_2(n^k) \geq n^2$, and thus equality holds. On the other hand, if $CS_2(n^k) = n^2$, then each pair of symbols must occur together precisely once in each pair of columns, and hence the covering suite must be an orthogonal array, which in turn implies the existence of $k-2$ mutually orthogonal Latin squares, and hence $k \leq n+1$. □

It is also well known that projective planes exist for all orders $n = p^\alpha$, which are powers of a single prime p. The construction of projective planes of prime power order was generalized by Bush [6] who proved the following result.

Theorem 3.2 Let $n = p^\alpha$ be a prime power with $n > t$. Then $CS_t(n^k) = n^t$ for all $k \leq n+1$. Moreover, if $n \geq 4$ is a power of 2, then $CS_3(n^k) = n^3$ for all $k \leq n+2$.

Proof. Let $F = \{0, 1, \ldots\}$ be the set of elements in a field of order n, with 0 being the zero element of the field. We index the columns of the orthogonal array by members of $F \cup \{\infty\}$, and the rows of the array are indexed by t-tuples $(\beta_0, \beta_1, \ldots, \beta_{t-1}) \in F^t$.

The array is then defined by the following table:

Column	Row	Array Entry
∞	$(\beta_0, \beta_1, ..., \beta_{t-1})$	β_0
0	$(\beta_0, \beta_1, ..., \beta_{t-1})$	β_{t-1}
$x \neq 0$	$(\beta_0, \beta_1, ..., \beta_{t-1})$	$\sum_{j=0}^{t-1} \beta_j x^j$

Figure 6. Construction of orthogonal arrays of strength t.

Let $T = (x_0, x_1, \ldots, x_{t-1})$ be a t-tuple of distinct columns, and let $B = (b_0, b_1, \ldots, b_{t-1})$ be an arbitrary member of F^t. To complete the proof of the first part of the theorem we need to show that B occurs as some row of the sub-matrix whose columns are indexed by T. If T contains neither ∞ nor 0, then we need to solve the following t equations for the t unknown quantities β_j, which index the row containing B.

$$\sum_{j=0}^{t-1} \beta_j {x_i}^j = b_i \quad \text{with} \quad 0 \leq i < t.$$

Now the coefficient matrix has the form of a Vandermonde matrix [43], and thus is invertible. Hence the system of equations has a unique solution. If T contains either ∞, or 0, or both, then we have a system of $t - 1$ or $t - 2$ equations that also have a Vandermonde coefficient matrix, and thus are uniquely solvable.

In the case where $t = 3$ and n is a power of two, we index the columns of the matrix by $F \cup \{\infty_0, \infty_1\}$ and we construct the array as follows:

Column	Row	Array Entry
∞_0	$(\beta_0, \beta_1, \beta_2)$	β_0
∞_1	$(\beta_0, \beta_1, \beta_2)$	β_1
0	$(\beta_0, \beta_1, \beta_2)$	β_2
$x \neq 0$	$(\beta_0, \beta_1, \beta_2)$	$\beta_0 + \beta_1 x + \beta_2 x^2$

Figure 7. Construction of orthogonal arrays of strength 3 over fields of characteristic 2.

The proof of this construction is similar, with the exception of the case in which we consider three columns indexed by two distinct non-zero field members, say x and y, and the column indexed by ∞_1. In this case we have to solve the following equations for β_0, β_1, and β_2.

$$\begin{aligned} \beta_0 + \beta_1 x + \beta_2 x^2 &= b_0 \\ \beta_0 + \beta_1 y + \beta_2 y^2 &= b_1 \\ \beta_1 &= b_2 \end{aligned}$$

which reduces to the following pair of equations in β_0 and β_2:

$$\begin{aligned} \beta_0 + \beta_2 x^2 &= b_0 - b_2 x \\ \beta_0 + \beta_2 y^2 &= b_1 - b_2 y \end{aligned}$$

Now these equations have a unique solution if and only if $x^2 - y^2 \neq 0$. In most fields this quantity may be zero for distinct values x and y, but in fields of characteristic 2, $x^2 - y^2 = (x - y)^2 \neq 0$. □

Remark 3.3 In the construction for arrays of strength 2, one can order the rows and columns so that the first q rows have the form $(0, x, x, \ldots, x)$ one for every member x of the field. This can be done by putting the 0 column on the left, and placing all rows where $\beta_1 = 0$ at the top of the array. (An example is shown in Figure 4.) Deleting these rows and the first column leaves us with an array with $q^2 - q$ rows, and q columns with the property that any ordered pair of distinct members of the field is contained in some row of any pair of columns. These structures are known in the literature as ordered designs (see Section IV.30 in [12]).We will use this construction in Section 7.

Bush [5] also gave the following product construction, which generalizes MacNeish's product construction for mutually orthogonal Latin squares.

Theorem 3.4 If there exist orthogonal arrays with k constraints, n_i levels (for $i = 1, 2$), strength t, and index 1, then there exists an orthogonal array with k constraints, $n_1 n_2$ levels, strength t, and index 1.

Proof. The construction is a straightforward product construction, indexing the rows of the new array by ordered pairs of indices of the input arrays, and using the Cartesian product of the symbol sets used as the symbol set of the new array. If the two input arrays are $A[i, j]$ and $B[m, j]$ then the resulting array $C[(i, m), j]$ is defined by $C[(i, m), j] = (A[i, j], B[m, j])$. The details are left as an exercise. □

Theorems 3.2 and 3.4 have the following consequences for covering suites:

Corollary 3.5 If $n = \prod q_j$ where the q_j are powers of distinct primes, then $CS_t(n^k) = n^t$, for any $k \leq 1 + \max(t, \min q_j)$.

There is a great deal of literature on the existence of sets of mutually orthogonal Latin squares; see [12] for an extensive list of references and numerical results. These results all have implications for the sizes $CS_2(n^k)$ of optimal pairwise covering suites with $k \leq n+1$. We quote two of the most famous of these results – the disproof of Euler's conjecture and the Chowla, Erdös, Straus Theorem. We will use these results in Section 7.

Theorem 3.6 (Bose, Parker, and Shrikhande [4]): For any positive integer n other than 2 or 6, there exists a pair of mutually orthogonal Latin squares of side n, and thus $CS_2(n^4) = n^2$ for all $n \notin \{2, 6\}$.

Euler originally conjectured (on the basis of Corollary 3.5) that no pair of mutually orthogonal Latin squares of side n exists for all $n \equiv 2(\text{mod}4)$. A proof of the non-existence of a pair of squares of side 6 was published in 1900, but Parker eventually found a pair of squares of side 10, and soon after, in 1960, the conjecture was totally discredited by Theorem 3.6.

Theorem 3.7 (Chowla, Erdös, Straus [10]) The number of mutually orthogonal Latin squares of side n goes to infinity with n, and for sufficiently large n, that number is at least $n^{0.0675}$.

In other words, if we fix k, then for all sufficiently large n, $CS_2(n^k) = n^2$. For small values of k much better results than that provided by Theorem 3.7 are known. For example $CS_2(n^4) = n^2$ for all $n > 6$. (Theorem 3.6), and $CS_2(n^5) = n^2$ for all $n > 10$ (see [12]).

One other result that belongs in this section is a construction due to Stevens, Ling, and Mendelsohn [39]. They give a construction for near optimal covering suites using affine geometries over finite fields. It is one of the few constructions in the literature for covering suites where not all the domains are of the same size.

Theorem 3.8 Let $n = p^\alpha$ be a prime power then $CS_2((n+1)^1, (n-1)^{n+1}) \leq n^2 - 1$.

The testing problem discussed in Section 1 is an application of the constructions described in this section. In that problem we wanted to test the pairwise interactions of a system with four parameters, two of cardinality 2, and two of cardinality 3. Now from Lemma 2.1 we have $CS_2(3^2, 2^2) \geq 9$. From the construction in Theorem 3.1 (illustrated in Figure 4) we have $CS_2(3^4) \leq 9$, and hence by the monotonicity results in Lemma 2.2, we can construct the test suite given in Figure 1, which illustrates that $CS_2(3^2, 2^2) \leq 9$. Thus the test suite in Figure 1 is optimal.

4. Lower Bounds and Asymptotics

Weak lower bounds on the function $CS_t(n^k)$ can be derived from non-existence theorems for orthogonal arrays. Two examples are presented below.

Theorem 4.1 (Bush:) An orthogonal array with k constraints, n levels, strength t, and index 1 exists only if:

$$k \leq \left\{ \begin{array}{lll} n+t-1 & \text{if } n \equiv 0(\text{mod}2) & \text{and} \quad t \leq n \\ n+t-2 & \text{if } n \equiv 1(\text{mod}2) & \text{and} \quad 3 \leq t \leq n \\ t+1 & \text{if} \quad t \geq n & \end{array} \right\}$$

This implies that $CS_t(n^k) > n^t$ for all k greater than the bounds given in Theorem 4.1.

Some tighter bounds on the existence of sets of mutually orthogonal Latin square due to Metsch [29] are very complicated to state (see II.2.23 in [12]) but have the following implications on the size of covering suites when $n \equiv 1, 2(\text{mod}4)$ and $n < 100$.

Corollary to Metsch's Theorem: $CS_2(n^k) > n^2$

(i) for all $k > n - 4$, when $n = 14, 21, 22$,
(ii) for all $k > n - 5$, when
$n = 30, 33, 38, 42, 46, 54, 57, 62, 66, 69, 70, 77, 78$, and
(iii) for all $k > n - 6$, when $n = 86, 93, 94$.

These lower bounds are of little use in that they do not tell us how fast $CS_t(n^k)$ grows as a function of k. The only case where tight lower bounds have been proved is when $n = t = 2$. This result has been rediscovered several times (see Rényi[35], Katona[24], Kleitman and Spencer[25], and Chandra, Kou, Markowsky, and Zaks [7] for examples).

Theorem 4.2 For all $k > 1$ we have $CS_2(2^k) = N$ where N is the smallest integer such that

$$k \leq \binom{N-1}{\lceil N/2 \rceil}$$

So we have the following table of exact values for $CS_2(2^k) = N$:

The proof of these lower bounds uses Sperner's lemma (see [22] for example) when N is even and the Erdös-Ko-Rado theorem [14] when N is odd.

k	2-3	4	5-10	11-15	16-35	36-56	57-126
N	4	5	6	7	8	9	10

Figure 8. The precise values of $N = CS_2(2^k)$.

The test suites that reach this bound may be constructed by taking the first test case to be the all 0 vector. The columns of the remaining $N - 1$ rows each contain precisely $\lceil N/2 \rceil$ ones, and each column is constructed by choosing a different $\lceil N/2 \rceil$-subset of the rows.

For example, when $n = t = 2$ and $k = 15$ we deduce that $N = 7$, and the optimal test suite is:

00000	00000	00000
11111	11111	00000
11111	10000	11110
11100	01110	11101
10011	01101	11011
01010	11011	10111
00101	10111	01111

Figure 9. An optimal pairwise covering suite for 15 binary parameters.

Gargano, Körner, and Vaccaro [20] established the following asymptotic results for the case of pairwise covering suites:

Theorem 4.3

$$\lim_{k\to\infty} CS_2(n^k)/\log k = n/2$$

Theorem 4.3 was considerably strengthened in a subsequent paper [19] by the same authors.

Let E be a set of ordered pairs $E \subset I_n^2$. We can define $CS_E(n^k)$ to be the minimum number of rows in an array with k columns over I_n, with the property that every pair of columns contains each of the members of E in at least one of its rows. Theorem 4.3 deals with the case in which $E = I_n^2$ but the result was strengthened in [19] and shown to hold for any subset E that contains a perfect matching. The implications of this result on the size of test suites is that one doesn't gain a great deal by excluding some of the pairwise interactions between parameters from consideration when constructing the test suites.

Stevens, Moura, and Mendelsohn [39] have also proved some other lower bounds on the sizes of covering suites with $t = 2$. Many of their results are too complicated to state here, but their improvements on Theorem 4.1 are easy to state, and provide useful information on small parameter sets.

Theorem 4.4 If $k \geq n + 2$, and $n \geq 3$ then $CS_2(n^k) \geq n^2 + 3$ with the only exception being $CS_2(3^5) = 11$.

We turn our attention now to lower bounds when $t = 3$. Kleitman and Spencer [25] proved that $CS_3(2^k) \geq 3.212 \ldots \log k(1 + o(1))$. I am not aware of any analogous results for $t > 3$.

The best asymptotic results on upper bounds for the covering suite numbers appear to come from coding theoretical constructions (e. g. Sloane [38]), and other deep results in probability theory and group theory.

A series of recent papers in the theoretical computer science community have been concerned with the de-randomization of randomized algorithms. As a by-product of this work, Naor and Naor [31], and Naor, Schulman and Srinvasan [32] have obtained results on binary covering suites ($n = 2$). Azar, Motwani, and Naor [1] have generalized these methods to apply to larger values of n. The best asymptotic upper bound given in these papers is:

Theorem 4.5 [32]: $CS_t(2^n) \leq 2^t t^{O(\log t)} \log n$

5. Greedy Covering Suites and Probabalistic Constructions

The first method that comes to a computer scientist's mind is to try to use a greedy algorithm to construct covering suites. The analysis of this algorithm gives a surprisingly good upper bound, but unfortunately, it is not a practical algorithm for the construction of good covering suites, since it has exponential complexity. The algorithm is as follows.

Let us define the *t-deficiency* $D_t(A)$ of a test suite A with k domains of size n to be the number of t-tuples of domain values not contained in any test case. Thus the deficiency of a covering suite is 0, whereas the deficiency of an empty set of test cases is

$$D(\phi) = \binom{k}{t} n^t$$

The greedy algorithm for the construction of a covering suite is to start with the empty test suite, $A_0 = \phi$, and at each stage to add a test case that decreases the deficiency by as much as possible. If A_S is the test suite after the choice of S test cases, then we will show that

$$D_t(A_S) \leq D_t(A_{S-1})(1 - n^{-t}) \tag{1}$$

Thus $D_t(A_S) \leq D_t(\phi)(1 - n^{-t})^S$. Let S be the smallest integer such that $D_t(A_S) < 1$ then $S = \left| -\log(D_t(\phi)/\log(1 - n^{-t}) \right|$ if we approximate $\log(1 - n^{-t})$ by $-n^{-t}$ we see that A_S is a covering suite when $S \approx n^t(\log \binom{k}{t} + t \log n)$ and hence we have:

Theorem 5.1 For all positive integers t, k, v, we have:

$$CS_t(n^k) \leq n^t(\log \binom{k}{t} + t \log n)$$

Proof. To establish equation (1) let us consider the incidence matrix with n^k rows – one for each possible test case – and $\binom{k}{t}n^t$ columns – one for each t-tuple to be covered (see Figure 10). Each row of the matrix contains $\binom{k}{t}$ ones, and each column contains n^{k-t} ones. Let $m = D_t(A_i) - D_t(A_{i-1})$ be the maximum number of ones in a row of the submatrix indexed by the deficient columns (i.e., the columns of t-tuples not yet covered by the array A_{i-1}). Counting the number of ones in this submatrix in two ways (ones per row times number of rows, and ones per column times number of columns) we obtain the inequality $mn^k \geq D_t(A_{i-1})n^{k-t}$, which implies equation (1). □

The major problem with this construction is that in order to compute the next test case, one must consider all $n^k - S$ possible test cases in order to choose one that decreases the deficiency as much as possible.

Godbole, Skipper, and Sunley [21] use probabilistic arguments to show that a randomly chosen array, in which each symbol is selected with equal probability, has a positive probability of being a covering suite if the number of rows is large enough. Their result is stated below in Theorem 5.2:

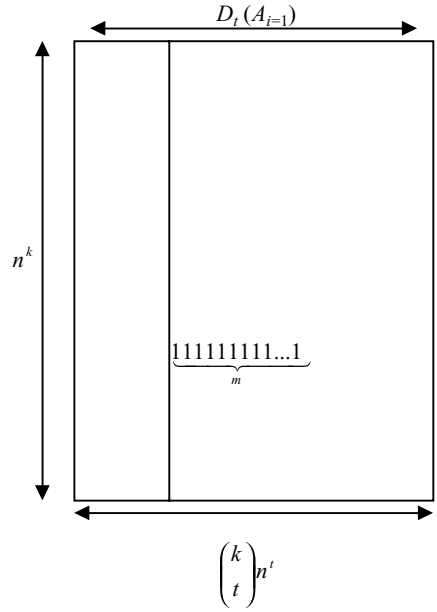


Figure 10. Proof of Equation (1).

Theorem 5.2 As $k \to \infty$,

$$CS_t(n^k) \leq \frac{(t-1)\log k}{\log(n^t/(n^t-1))}\{1+o(1)\}$$

Both Sloane[38] and Godbole et. al. [21] also prove the following stronger result due to Roux [36] for the case in which $t = 3$ and $n = 2$. The bound is stronger because the construction considers only the selection of columns with an equal number of zeros and ones, rather than selecting at random each entry in the matrix with a probability of $1/2$.

Theorem 5.3 As $k \to \infty$,

$$CS_3(2^k) \leq 7.56444 \cdots \log k\{1+o(1)\}$$

6. Algebraic Constructions

Two papers by Chateauneuf, Colbourn and Kreher [9] and Chateauneuf and Kreher [8] illustrate methods of using algebraic techniques to construct covering suites.

Let S be an $N \times k$ array over the set I_n. Let Γ be a subgroup of the group of all permutations of the symbols in I_n, and let $m = |\Gamma|$. For $g \in \Gamma$, we define the image Sg of S to be the array whose entries are the images of the corresponding members of S under the action of the permutation g. We further define the image S^Γ of S under Γ to be the $mN \times k$ array consisting of the rows of Sg for all $g \in \Gamma$.

For example, let $\Gamma = \{(0)(1)(2), (012), (021)\}$ be the cyclic group of permutations of I_3, and let

$$S = \begin{matrix} 000 \\ 012 \\ 021 \end{matrix}$$

Then S^Γ has nine rows, the first three being the image of S under (0)(1)(2), the identity permutation; the next three are the image of S under (012); and the last three are the image of S under (021).

$$S^\Gamma = \begin{matrix} 000 \\ 012 \\ 021 \\ 111 \\ 120 \\ 102 \\ 222 \\ 201 \\ 210 \end{matrix}$$

The array S is called a starter array with respect to Γ if every $n \times t$ subarray contains at least one representative of each orbit of Γ acting on ordered t-tuples from I_n.

In the previous example, there are three orbits of ordered pairs under Γ — the orbit of 00, which contains the pairs $\{00, 11, 22\}$, the orbit of 01, which contains the pairs $\{01, 12, 20\}$ and the orbit of 02, which contains the pairs $\{02, 10, 21\}$. The array S is a starter array with respect to Γ since every 3×2 subarray contains a representative of each orbit of ordered pairs, and thus S^Γ is a $CS_2(3^3)$.

Note that we may also reduce the number of rows in S^Γ by deleting a copy of any row that is repeated.

For example, with $t = n = 3$, and Γ is the symmetric group of all six permutations of I_3, there are five orbits of ordered triples, namely the orbits of 000, 001, 010, 011, and 012. The array

$$S = \begin{array}{c} 0000 \\ 0012 \\ 0101 \\ 0110 \\ 0122 \end{array}$$

contains representatives of each of the five orbits in each of the four 3×5 subarrays. Taking the image of S under the symmetric group and removing the duplicate images of the first row generates an array with 27 rows, which establishes that $CS_3(3^4) = 27$ (c.f., Theorem 3.2).

The main constructions given in [9] involve the use of the projective general linear group $PGL(q)$ defined as a group of permutations of the projective line $GF(q) \cup \{\infty\}$, where q is a prime power and $GF(q)$ is the Galois field of order q. The projective linear group consists of all permutations of $GF(q) \cup \{\infty\}$ in the set

$$PGL(q) = \left\{ x \mapsto \frac{ax + b}{cx + d} : a, b, c, d \in GF(q), ad - bc \neq 0 \right\}$$

where we define $1/0 = \infty, 1/\infty = 0, \infty + 1 = 1 - \infty = 1 \times \infty = \infty$, and $\infty/\infty = 1$.

The size of the group is $q^3 - q$. The action of this group on the projective line is sharply 3-transitive, meaning that there is a unique member of the group that takes any ordered triple of distinct elements to any other ordered triple of distinct elements. This means that if an array S over $GF(q) \cup \{\infty\}$ has the property that any $N \times 3$ subarray contains the five rows with patterns (xxx), (xxy), (xyx), (xyy), and (xyz), then the image of S with duplicates removed is a covering suite of strength 3.

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	2	2	1	0
1	1	0	1	2	2	0
2	2	1	0	1	2	0
3	2	2	1	0	1	0
4	1	2	2	1	0	0

Figure 11. A starter array for $PGL(3)$.

Chateauneuf et al. [9] then give an ingenious construction of an array with $2n-1$ rows and $2n$ columns over I_n with the property that every $(2n-1)\times 3$ sub-array contains representatives of the four non-constant orbits of triples (xxy), (xyx), (xyy), and (xyz). Taking the image of this array under $PGL(q)$, where $q=n-1$, together with the n constant rows of length, $2n$ gives a covering suite of strength 3. A special case of their construction of the starter array is given by indexing the rows by I_{2n-1}, the columns by I_{2n}, and setting $S[i,2n-1]=0$ and $S[i,j]=\min\{|i-j|, 2n-1-|i-j|\}$ for all $i,j\in I_{2n-1}$. This array S has the property that for any two columns j_1 and j_2 there is precisely one row i, such that $S[i,j_1]=S[i,j_2]$ and, furthermore, all other entries in that row are different from the common value. The exact value of i is given by solving the equation $2i=j_1+j_2(\mathrm{mod}2n-1)$ or $i=j_1$ in the special case that $j_2=2n-1$. We give an example of this construction in Figure 11.

This property also ensures that given any three columns, there are precisely three rows in which not all of the entries in the row are distinct. Thus, so long as $2n-1>3$, we have a representative of the orbit of (xyz) in some row. Hence, we have the following:

Theorem 6.1 Let q be a prime power, then

$$CS_3((q+1)^{2(q+1)})\leq (2q+1)(q^3-q)+q+1$$

A consequence of this result is that $CS_3(3^6)=33$. The theorem provides the upper bound; the lower bound is a consequence of Lemma 2.3 and the result that $CS_2(3^5)=11$, see [38].

There is another algebraic starter construction in [9] which establishes that $CS_3(4^8)\leq 88$.

7. Recursive Constructions

A recursive construction for a covering suite is a method for constructing a covering suite from one or more covering suites with smaller parameter sets. We begin with an efficient recursive construction for pairwise covering suites. This construction was probably known to Cohen et al. [11] – since the array they construct to show that $CS_2(3^{13}) \leq 15$ has the form of the construction. However, it first appeared in full detail in Williams' paper [45].

Theorem 7.1 If q is prime power, then
$CS_2(q^{kq+1}) \leq CS_2(q^k) + q^2 - q$.

Proof. Let A be a pair-wise covering test suite with k columns and $N = CS_2(q^k)$ rows with entries from I_q.

Construct a new array B with N rows and $kq + 1$ columns by taking each column A^i of Aq times and bordering it with an additional column of zeroes. The additional $q^2 - q$ rows are constructed by taking the last rows from the orthogonal array C constructed in the proof of Theorem 3.2 and Remark 3.3, taking the first column once, and the remaining q columns k times each (see Figure 12). □

Corollary 7.2 If q is prime power, and d is any positive integer, then $CS_2(q^{q^d + q^{d-1} + \cdots + q + 1}) \leq dq^2 - (d-1)q$.

Proof. The result follows from Theorem 3.2 when $d = 1$, and by induction on d using Theorem 7.1. □

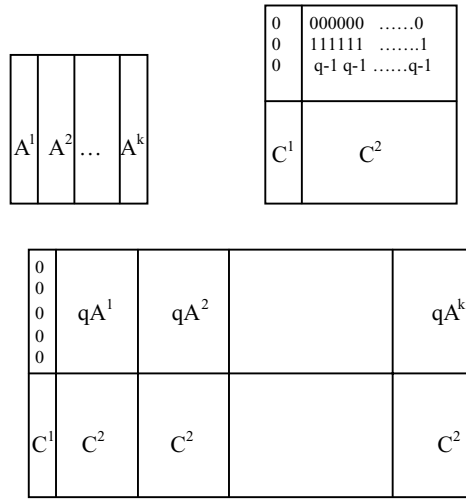


Figure 12. Construction for Theorem 7.1.

This result implies a good constructive upper bound on the size of pairwise covering test suites.

Theorem 7.3 There is an absolute constant C such that $CS_2(n^k) \leq Cn^2 \log k$ for all positive integers k and n.

Proof. By Bertrand's postulate (proved by Chebyshev in 1851) there is a prime p, between n and $2n$. In fact, for $n > 115$, there is always a prime between n and $1.1n$ (see [23]). Let d be the smallest integer such that $k \leq 1 + p + p^2 + \cdots + p^d$.

This implies that $1 + p + p^2 + \cdots + p^{d-1} < k$, and hence that $d = O(\log k)$. Now applying Corollary 7.2 and the monotonicity result, Lemma 2.2, we have:

$CS_2(n^k) \leq CS_2(p^{1+p+\cdots+p^d}) \leq dp^2 = O(n^2 \log k)$ thus proving the theorem. □

Another recursive construction, which has been rediscovered many times is the following result, which gives a method of squaring the number k of parameters in a covering suite of strength t while multiplying the number of test cases by a factor dependent only on t and n, but independent of k. This factor is related to the Turan numbers $T(t, n)$(see [44]) that are defined to be the number of edges in the Turan graph. The Turan graph is the complete n-partite graph with t vertices, having b parts of size $a + 1$, and $n - b$ parts of size $a = \lfloor t/n \rfloor$ where $b = t - na$. Turan's theorem (1941) states that among all t-vertex graphs with no $n + 1$ cliques, the Turan graph is the one with the most edges.

Note that when $n \geq t$, $T(t, n) = t(t - 1)/2$, and that when $n = 2$, we have $T(t, 2) = \lfloor t^2/4 \rfloor$.

Theorem 7.4 If $CS_t(n^k) = N$ and there exist $T(t, n) - 1$ mutually orthogonal Latin squares of side k (or equivalently $CS_2(k^{T(t,n)+1}) = k^2$) then $CS_t(n^{k^2}) \leq N(T(t, n) + 1)$.

Before proving this result, we note that this generalizes Tang and Chen's result [41], since they require k to be a prime. It generalizes and strengthens the result of Chateuneuf et al. [9] by removing the divisibility conditions on k, and producing smaller arrays in the cases where $n < t$.

Proof. Let A be a t-wise covering test suite with k columns, N rows, entries from I_n, and let A^i be the i-th column of A. Let $B = B[i, j]$ be an orthogonal array of strength 2 with $T(t, n) + 1$ columns and entries from $\{1, 2, \ldots, k\}$. We will construct a block array C with k^2 columns and $T(t, n) + 1$ rows. Each element in C will be a column of A. Let $A^{B[j,i]}$ be the block in the i-th row and j-th column of C (see Figure 13).

Now consider T, an arbitrary t-tuple of members of I_n. Let C' be a submatrix of C induced by an arbitrary choice of t columns. We wish to show that T is a row of C'.

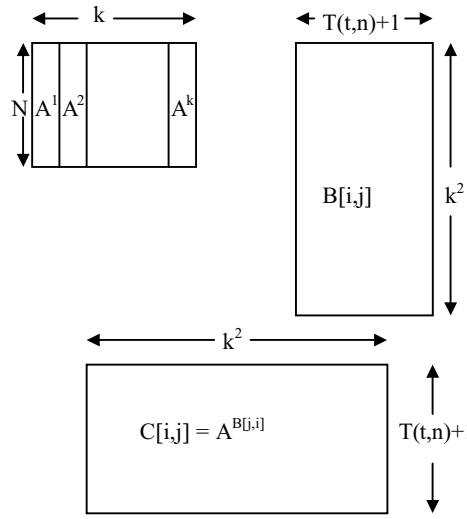


Figure 13. The construction for Theorem 7.4.

The columns of C' correspond to t rows of B. Let B' be the submatrix of B induced by those t rows. We wish to find a column in B' with distinct values whenever T has distinct values in the corresponding coordinates, since this would guarantee that T is a row in C' using the properties of the base array A.

Since B is an orthogonal array, whenever $B[i, j] = B[k, j]$ then $B[i, m] \neq B[k, m]$ for every column $m \neq j$. This means that any pair of distinct values in T eliminates at most one column of B'. By Turan's theorem, the number of pairs of distinct values in T is at most $T(t, n)$, and hence at least one column in B' contains distinct values whenever T contains distinct values. This completes the proof. □

The result has several immediate corollaries.

Corollary 7.5 (Boroday [1]): $CS_3(2^{k^2}) \leq 3CS_3(2^k)$.

Proof. The result follows from Theorem 7.4 since $T(3, 2) = 2$ and $CS_2(k^3) = k^2$ for all k, by Corollary 3.5. □

Using the disproof of Euler's conjecture we can derive:

Corollary 7.6 For all $k > 2$, $k \neq 6$, $n > 2$ we have $CS_3(n^{k^2}) \leq 4CS_3(n^k)$.

Proof. This result also follows from Theorem 7.4 since $T(3, n) = 3$ for all $n > 2$, and $CS_2(k^4) = k^2$ for all $k > 2$, $k \neq 6$, by Theorem 3.6. □

We also derive one more corollary using the result of Chowla, Erdös, and Strauss:

Corollary 7.7 For all positive integers t and n, and all sufficiently large integers k, we have $CS_t(n^{k^2}) \leq (T(t,n)+1)CS_t(n^k)$.

We give one more recursive result that shows how to double the number of columns in 3-wise and 4-wise covering suites. The result for $t = 3$ and $n = 2$ was first proved by Roux [36]. The result for $t = 3$ appears in [9], although our proof is different. To the best of my knowledge, the result for $t = 4$ is new.

A tool used in the construction is a partition of the edge set of the complete directed graph on n vertices, such that each part contains a spanning set of directed cycles – i.e., each vertex occurs precisely once as the head of an arc and once as the tail of an arc. The simplest such partition is given by the following construction:

$$F_j = \{(i, i + j(\mathrm{mod}\, n)) : i \in I_n\},\ j = 1, 2, \ldots, n-1$$

It is clear from the construction that each vertex appears precisely once as the head of an arc in F_j, and precisely once as the tail of some arc. To show that each arc in the complete directed graph occurs in precisely one of the sets F_j, consider the arc (x, y) and note that it occurs in the set F_{y-x}.

We now proceed to use this construction as an ingredient in the following doubling constructions for strength 3 and 4 covering suites.

Theorem 7.8 For all positive integers n and k,

a) $CS_3(n^{2k}) \leq CS_3(n^k) + (n-1)CS_2(n^k)$
b) $CS_4(n^{2k}) \leq CS_4(n^k) + (n-1)CS_3(n^k) + CS_2((n^2)^k)$

Proof. a) Let A be a 3-wise covering suite with k columns and $CS_3(n^k)$ rows over the symbol set I_n. Construct a new array by taking each column of A twice and adding $(n-1)CS_2(n^k)$ rows constructed as follows. Let B be a pairwise covering suite with k columns and $CS_2(n^k)$ rows over the symbol set I_n. Take $n-1$ copies of B, and replace the i-th symbol in the j-th copy with the i-th member (an ordered pair) of F_j. A worked example of this construction is given below.

To verify that this construction yields a 3-wise covering suite, we need to verify that three types of triples are covered by some row of the array: triples with three elements in distinct columns of the original array A, triples with two equal symbols in columns that came from the same column of A, and triples with two unequal symbols in columns that came from the same column in A. The first and second types of triples are covered due to the original structure of A, and the third type of triple is covered by the construction of the additional rows from B.

The construction and proof for b) is similar. Let A, B, and C be 4-wise, 3-wise, and pairwise covering suites with the parameters given in the statement of the result. Take each column of A twice, take $n-1$ copies of B and replace the i-th symbol in

$$\begin{array}{lll} 0000 & \rightarrow & 00\ 00\ 00\ 00 \\ 0012 & \rightarrow & 00\ 00\ 11\ 22 \\ 0021 & \rightarrow & 00\ 00\ 22\ 11 \\ & \cdots & \\ 2212 & \rightarrow & 22\ 22\ 11\ 22 \\ 2220 & \rightarrow & 22\ 22\ 22\ 00 \end{array}$$

Figure 14. The doubling process in the construction of Theorem 7.5.

the j-th copy with the i-th member of F_j, then take a copy of C, and replace the i-th symbol with the i-th member of $I_n \times I_n$ in some arbitrary ordering of the members of this Cartesian product. □

Example 7.6 We now illustrate the use of Theorem 7.5 to show that:

$$CS_3(3^8) \leq 45 = 27 + 9 + 9$$

The first 27 rows of the array come from doubling each column of the array constructed in Theorem 3.2, which shows that $CS_3(3^4) = 27$.

The next 9 rows come from substituting the members of $F_1 = \{01, 12, 20\}$ for the three symbols in Figure 4, which is a minimal $CS_2(3^4)$, see Figure 15.

The final 9 rows come from substituting the members of $F_2 = \{02, 10, 21\}$ in the same array. □

Results similar to those of Theorem 7.5 are probably true for higher values of t, but the number of cases in the proof detracts from the aesthetics of such results.

8. Heuristics

In this section, we discuss how the techniques presented in the previous sections can be used and extended by heuristic methods to solve practical problems in the generation of covering suites.

$$\begin{array}{lll} 0000 & \rightarrow & 01\ 01\ 01\ 01 \\ 0111 & \rightarrow & 01\ 12\ 12\ 12 \\ 0222 & \rightarrow & 01\ 20\ 20\ 20 \\ 1021 & \rightarrow & 12\ 01\ 20\ 12 \\ 1102 & \rightarrow & 12\ 12\ 01\ 20 \\ 1210 & \rightarrow & 12\ 20\ 12\ 01 \\ 2012 & \rightarrow & 20\ 01\ 12\ 20 \\ 2120 & \rightarrow & 20\ 12\ 20\ 01 \\ 2201 & \rightarrow & 20\ 20\ 01\ 12 \end{array}$$

Figure 15. The substitution process in the construction of Theorem 7.5.

The most common problem, and the one on which we have focused so far, is that of generating a minimal set of test cases guaranteeing t-wise coverage of k parameters with domains of sizes $n_1, n_2, \ldots, n_k$. The practical issues of limited time and space require that we should find a polynomial time algorithm to solve this problem. Lei and Tai [26] prove that the determination of $CS_2(n^k)$ is NP-complete using a reduction to the vertex cover problem. Seroussi and Bshouty [37] prove that the determination of $CS_t(2^k)$ is NP-complete using a reduction to graph 3-coloring. Thus, it seems unlikely that we will find a polynomial time algorithm for constructing minimal covering suites in the general case.

Another interesting and practical problem is that of finding a test suite with minimal deficiency, given a fixed budget for executing a maximum of N test cases. This problem is theoretically equivalent to the problem of finding a minimal test suite, so it, too, is NP-complete.

Yet a third problem is that of finding a minimal test suite with a fixed *relative deficiency*, where the relative deficiency is defined as the deficiency divided by the total number of t-subsets to be covered. In the case where all domains are the same size, the relative deficiency RD_t of a test suite A is defined as:

$$RD_t(A) = \frac{D_t(A)}{n^t \binom{k}{t}}$$

A surprising result from Roux's thesis [36] states that for any t and any $\varepsilon > 0$, there is a constant $N(t, \varepsilon)$, independent of k, such that there exists a test suite A with $N(t, \varepsilon)$ test cases for k binary parameters with relative deficiency $\varepsilon = RD_t(A)$. Sloane's paper [38] quotes the result that $N(3,0.001) \leq 68$. Unfortunately, the arguments are probabilistic, and they do not give a deterministic algorithm for finding such a test suite.

Lei and Tai [26] also discuss the practical issue of extending a given test suite. Assuming that a pairwise covering test suite for k parameters is already given, what is the best way to add a single column, and perhaps additional rows, in order to extend this suite for the additional parameter? They give an optimal algorithm for adding new rows once a single column has been added to the initial test suite. However, their algorithms for adding a new column are either exponential or sub-optimal.

Our heuristics for solving these problems are a combination of the constructive and recursive methods given in sections 3 and 7, probabilistic algorithms inspired by Roux's techniques, and a greedy heuristic for the completion of partial test suites. We also use the monotonicity results when the domain sizes are inappropriate for the finite field methods of Section 3.

Roux's technique is particularly appropriate in the case of a fixed testing budget N. To apply the technique, we generate k random sets of columns of length N, with each symbol appearing either $\lfloor N/n_i \rfloor$ or $\lceil N/n_i \rceil$ times in the column. We then select one

column from each of these sets in such a way as to minimize the deficiency of the array that we generate. We use a greedy heuristic for the minimization, since the selection of columns is reducible to the problem of finding a maximal clique in a graph.

Our heuristic for the completion of a partial test suite is different from that given by Cohen, Dalal, Fredman, and Patton in [11]. Assume that we are given a partial test suite, and we are required to add a new test case. We first find the set of t columns with the largest number of missing t-tuples, and select one of the missing tuples as the values in those columns. We then rank all the remaining (column, value) pairs by computing t values, $(p_0, p_1, \ldots p_{t-1})$—which we call the *potential vector*. The first of these values p_0 is the amount by which the inclusion of the ranked value in the ranked column in the partial test case would decrease the deficiency. In other words, p_0 counts the number of t-tuples containing the value in the column and $t - 1$ other values that have already been fixed in the partial test case under construction. In general, p_i counts the total number of missing t-tuples containing that value in that column as well as $t - 1 - i$ values that have already been fixed, and i undecided values in the other columns. We then choose the (column, value) pair with the lexicographically maximum potential vector. If several pairs achieve the same maximum potential vector, we break the tie by a random choice among those pairs that achieve the maximum.

For small values of t, k, and n, Nurmela [33] has used the method of tabu search effectively to find small covering suites, and some of the smallest known suites according to the tables in [9] are due to this technique.

9. Applications

In the introduction we discussed the application of covering suites to the testing of a software or hardware interface with k parameters. In this section we discuss three other applications of covering suites: two in the area of software and hardware testing, and one in the seemingly unrelated area of search algorithms.

9.1. Reducing State Machine Models

It is common in both the hardware and software domains to specify the behavior of a unit to be tested by a state machine or transition system. A full account of this area may be found in [34]. The states represent the possible states for a software unit (e.g., the screen currently displayed, and the fields and actions currently active or disabled). The transitions between states are arcs labeled by the input stimulus from the user that cause the unit to change state. A great deal of research activity has been devoted to the analysis of these state machine models, but most of this effort is stymied by the so-called state explosion problem. The size of the state machine grows exponentially, even when the unit under test has a relatively simple structure.

In [16] Farchi, Hartman, and Pinter describe the use of a state machine exploration tool to generate test cases for the testing of implementations of software standards. In

this context, a test case is not merely a k-tuple of input values, but it is a sequence of input stimuli, each of which is a tuple of input values. In graph theoretical terms, it is a path or walk through the arcs of the graph of the state machine.

The test generation tool described in [16] is built on the basis of the Murφ model checker [13]. The Murφ system builds a labeled directed graph to describe all possible behaviors of the system under test. The number of arcs leaving any particular state is equal to the number of possible stimuli which the tester can apply to the system at that state. Since the graph describes all possible behaviors, the number of arcs leaving any state is equal to the cardinality of the Cartesian product of the domains of all the input parameters. If instead of building an arc for each member of the Cartesian product, we use a covering suite subset of those arcs, we obtain a much smaller state machine. The new state machine does not describe all possible behaviors of the system under test, but it provides a good sample of the full set of behaviors. In model checking it is vital to describe the entire state machine, but in the context of test generation it is sufficient to sample the behavior of the system under test.

An example of the use of this technique is provided by the model we used to test the file system described in [16]. The model of the file system contains two users who interact with the system by sending requests to read, write, open, or close a file. The open and close commands do not have parameters, but the read and write commands each have five parameters in the model, two with domains of size 4, two of size 3, and one binary parameter. The states of the model may be divided into three classes, states where neither user may issue a read or write command, states where only one of the users may issue a read or write command, and states where both users may issue these commands.

In the full Cartesian product model, each state of the second class has $288 = 4 \times 4 \times 3 \times 3 \times 2$ "read" arcs leaving it, and the same number of "write" arcs. States of the third class have twice that many arcs leaving them since both users can perform both reads and writes. We can create a covering suite with only 16 rows which will cover all pairwise interactions between the parameters of the read and write commands. This reduces the number of arcs leaving a state of the third class from $1152 = 288 \times 4$ to $64 = 16 \times 4$.

The resulting state machine has fewer reachable states, and thus does not describe the full range of file system behavior, but it is adequate for generating a comprehensive test suite which ensures compliance to the software standard.

9.2. Path Coverage Using Covering Suites

A further application of covering arrays in the context of state machine models is their use as a means of achieving a reduced form of path coverage of the model.

When a state machine model is used to generate test cases for an application, the most common criteria for generating the test suite is the achievement of a coverage

goal. A suite of test cases that passes once through every state is said to achieve *state coverage.* In graph theory terms, this resembles a path partition of the underlying directed graph. In conformance testing it is common practice to generate an Euler tour of the transition system to guarantee transition coverage. A stronger form of coverage requires the coverage of all paths of length two or greater.

We have used covering suites to define a test generation strategy that gives a moderate sized test suite with interesting path coverage properties. Recall that in the state machine, each arc is labeled by the stimulus that causes the unit under test to change state. We can view test cases with k transitions as sequences of arc labels, and if we select sequences of arc labels from a covering suite of strength 2, we achieve a level of coverage guaranteeing that all pairs of transition types are tested at all distances of up to k apart.

In the file system example discussed in the previous section we may wish to generate a series of test cases each containing 10 read commands, alternating between the two users. Having reduced the number of possible read commands to 16, we are still faced with the prospect that there are 16^{10} possible test cases with 10 steps. Again we can restrict ourselves to a set of sequences of length 10 taken from a pairwise covering suite, and with these 256 test cases we guarantee that all of the pairwise interactions between the various read commands have been tested together in the same test sequence.

9.3. Blind Dyslectic Synchronized Robots on a Line

A final application of covering suites, in a completely different area was pointed out to us by S. Gal.

Lim and Alpern [27] have studied the minimax rendezvous time problem for k distinguishable robots on a line. The robots are initially placed at the points $1, 2, \ldots, k$ in some permutation. The robots are dyslectic in the sense that they do not have a common notion of the positive direction on the line. The robots cannot see each other and they only become aware of each other's presence by the sense of touch. The minimax rendezvous time is the minimum over all strategies of the maximum over all possible starting configurations of the time by which they can all get together. All the robots move at the same speed and are synchronized to start their rendezvous strategy together.

Gal[18] has exhibited a simple strategy where, in the first phase, the robots at positions 1 and k identify the fact that they are at the extreme positions, and then, in a second phase proceed toward the center, gathering all their colleagues, who remain passive in the second phase. In Gal's algorithm the first phase takes $2(1 + \lceil \log_2 k \rceil)$ steps, and the second phase takes $\lceil k/2 \rceil$ steps. We propose a variant on Gal's algorithm, which decreases the number of steps required in the first phase.

We provide each robot with a binary sequence of length $N = CS_2(2^k)$. The sequence for the k-th robot is the k-th column of an optimal covering suite. The robot interprets a 0 as an instruction to move half a unit to its left, then half a unit right; a 1 is interpreted as a move half right then half left. Since the robots do not have a common sense of direction, in order to guarantee that each internal robot meets both its neighbors, the pair of robots must execute the four patterns 00, 01, 10, and 11. After precisely N time units, all the internal robots have met both their neighbors, and thus the end robots can be sure that they are on the ends, and begin the second phase. Theorem 4.2 guarantees that $N < 1 + \lceil \log_2 k \rceil$.

Further Reading

In this section we give a series of urls for web based resources relevant to the subjects discussed in this paper.

Thomas Huckle has a very interesting collection of software bugs at http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html

Peter Cameron's "Design Resources on the Web": http://www.maths.qmw.ac.uk/~pjc/design/resources.html

Neal Sloane's library of orthogonal arrays is at : http://www.research.att.com/~njas/oadir/index.html

The Handbook of Combinatorial Designs [12] has its own web page which is regularly updated with new results: http://www.emba.uvm.edu/~dinitz/hcd.html

Telecordia's web-based service for the generation of test cases is at: http://aetgweb.argreenhouse.com/

Harry Robinson's Model-based testing website has many interesting resources: http://www.geocities.com/model_based_testing/

A search at http://citeseer.nj.nec.com with the keyword "derandomization" yields a wealth of material on the asymptotic results quoted in Section 4.

Acknowledgments

I would like to thank Marty Golumbic, Cindy Eisner, and Shmuel Gal for reading earlier drafts of this paper and providing helpful insights. I would like to thank Leonid Raskin who assisted in the implementation of the constructive algorithms and heuristics. I would also like to thank Gyorgy Turan and Seffi Naor for their help with the recent work on the de-randomization of algorithms. I would also like to thank Serge Boroday for his helpful correspondence, and Brett Stevens for his careful reading of the paper and for pointing out an error in an earlier proof of Theorem 7.8.

References

[1] Azar J., Motwani R., and Naor J., Approximating probability distributions using small sample spaces. *Combinatorica* 18: 151–171 (1998).

[2] Boroday S. Y., Determining essential arguments of Boolean functions. (in Russian), in Proceedings of the Conference on Industrial Mathematics, Taganrog (1998), 59–61. The translation is a personal communication from the author.

[3] Boroday S. Y. and Grunskii I. S., Recursive generation of locally complete tests. *Cybernetics and Systems Analysis* 28: 20–25 (1992).

[4] Bose R. C., Parker E. T., and Shrikhande S., Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture. *Can. J. Math.* 12: 189–203 (1960).

[5] Bush K. A., A generalization of the theorem due to MacNeish. *Ann. Math. Stat.* 23: 293–295 (1952).

[6] Bush K. A., Orthogonal arrays of index unity. *Annals of Mathematical Statistics* 23: 426–434 (1952).

[7] Chandra A. K., Kou L. T., Markowsky G., and Zaks S., On sets of Boolean n-vectors with all k-projections surjective. *Acta Inform.* 20: 103–111 (1983).

[8] Chateauneuf M. A. and Kreher D. L., On the state of strength 3 covering arrays. J. Combinatorial Designs, *Journal of Combinatorial Designs* 10 (2002), no. 4, 217–238. Available at http://www.math.mtu.edu/∼kreher/.

[9] Chateauneuf M. A., Colbourn C. J., and Kreher D. L., Covering arrays of strength 3. *Designs, Codes, and Cryptography*, 16: 235–242 (1999).

[10] Chowla S., Erdös P., and Straus E. G., On the maximal number of pairwise orthogonal Latin squares of a given order. *Canad. J. Math.* 13: 204–208 (1960).

[11] Cohen D. M., Dalal S. R., Fredman M. L., and Patton G. C., The AETG System: An approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23: 437–444 (1997).

[12] Colbourn C. J. and Dinitz J. H., The CRC Handbook of Combinatorial Designs. *CRC Press* (1996).

[13] Dill D., Murϕ Description Language and Verifier. http://sprout.stanford.edu/dill/murphi.html.

[14] Edelman A., The mathematics of the Pentium division bug. *SIAM Review* 39: 54–67 (1997).

[15] Erdös P., Ko C., and Rado R., Intersection theorems for systems of finite sets. *Quart. J. Math. Oxford* 12: 313–318 (1961).

[16] Farchi E., Hartman A., and Pinter S. S., Using a model-based test generator to test for standards conformance. *IBM Systems Journal* 41: 89–110 (2002).

[17] Fisher R. A., An examination of the different possible solutions of a problem in incomplete blocks. *Ann. Eugenics* 10: 52–75 (1940).

[18] Gal S., Rendezvous search on a line. *Operations Research* 47: 974–976 (1999).

[19] Gargano L., Körner J., and Vaccaro U., Capacities: from information theory to extremal set theory. *J. Comb. Theory, Ser. A.* 68: 296–316 (1994).

[20] Gargano L., Körner J., and Vaccaro U., Sperner capacities. *Graphs and Combinatorics*, 9: 31–46 (1993).

[21] Godbole A. P., Skipper D. E., and Sunley R. A., t-Covering arrays: Upper bounds and Poisson approximations. *Combinatorics, Probability, and Computing* 5: 105–117 (1996).

[22] Greene C., Sperner families and partitions of a partially ordered set. In Combinatorics (ed. M. Hall Jr. and J. van Lint) Dordrecht, Holland, (1975) pp. 277–290.

[23] Harborth H. and Kemnitz A., Calculations for Bertrand's Postulate. *Math. Magazine* 54: 33–34 (1981).

[24] Katona G. O. H., Two applications (for search theory and truth functions) of Sperner type theorems. *Periodica Math. Hung.* 3: 19–26 (1973).

[25] Kleitman D. J. and Spencer J., Families of k-independent sets. *Discrete Math*. 6: 255–262 (1973).

[26] Lei Y. and Tai K. C., In-parameter order: A test generation strategy for pairwise testing. in Proc. 3rd IEEE High Assurance Systems Engineering Symposium, (1998) pp. 254–161.

[27] Leveson N. and Turner C. S. An investigation of the Therac-25 accidents. *IEEE Computer*, 26: 18–41 (1993).

[28] Lim W. S. and Alpern S., Minimax rendezvous on the line. *SIAM J. Control and Optim.* 34: 1650–1665 (1996).

[29] Lions J. L., Ariane 5 Flight 501 failure, Report by the Inquiry Board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[30] Metsch K., Improvement of Bruck's completion theorem. *Designs, Codes and Cryptography* 1: 99–116 (1991).

[31] Naor J. and Naor M., Small-bias probability spaces: efficient constructions and applications. *SIAM J. Computing*, 22: 838–856 (1993).

[32] Naor M., Schulman L. J., and Srinvasan A., Splitters and near-optimal randomization. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, (1996), pp. 182–191.

[33] Nurmela K., Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138: 143–152 (2004).

[34] Peled D. A., Software reliability methods. Springer, New York (2001).

[35] Rényi A., Foundations of probability. Wiley, New York, (1971).

[36] Roux G., *k-propriétés dans les tableaux de n colonnes; cas particulier de la k-surjectivité et de la k-permutativité*. Ph. D. Dissertation, University of Paris 6, (1987).

[37] Seroussi G. and Bshouty N. H., Vector sets for exhaustive testing of logic circuits. *IEEE Trans. Information Theory*, 34: 513–522 (1988).

[38] Sloane N. J. A., Covering arrays and intersecting codes. *J. Combinatorial Designs* 1: 51–63 (1993).

[39] Stevens B., Ling A., and Mendelsohn E., A direct construction of transversal covers using group divisible designs. *Ars Combin.* 63: 145–159 (2002).

[40] Stevens B., Moura L., and Mendelsohn E., Lower bounds for transversal covers. *Designs, Codes, and Cryptography*, 15: 279–299 (1998).

[41] Tang D. T. and Chen C. L., Iterative exhaustive pattern generation for logic testing. *IBM J. Res. Develop.* 28: 212–219 (1984).

[42] Tang D. T. and Woo L. S., Exhaustive test pattern generation with constant weight vectors. *IEEE Trans. Computers* 32: 1145–1150 (1983).

[43] Web W. W., The Vandermonde determinant, http://www.webeq.com/WebEQ/2.3/docs/tour/determinant.html

[44] West D. B., Introduction to Graph Theory. *Prentice Hall NJ,* (1996).

[45] Williams A. W., Determination of Test Configurations for Pair-Wise Interaction Coverage. *in Proceedings of the 13th International Conference on the Testing of Communicating Systems (TestCom 2000)*, Ottawa Canada, (2000) pp. 59–74.

11
Incidences

János Pach
Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012, USA

Micha Sharir
School of Computer Science,
Tel Aviv University, Tel Aviv 69978, Israel and
Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012, USA

Abstract

We survey recent progress related to the following general problem in combinatorial geometry: What is the maximum number of incidences between m *points and* n *members taken from a fixed family of curves or surfaces in* d*-space? Results of this kind have found numerous applications to geometric problems related to the distribution of distances among points, to questions in additive number theory, in analysis, and in computational geometry.*

1. Introduction

The problem and its relatives. Let P be a set of m distinct points, and let L be a set of n distinct lines in the plane. Let $I(P, L)$ denote the number of *incidences* between the points of P and the lines of L, i.e.,

$$I(P, L) = |\{(p, \ell)|\ p \in P, \ell \in L, p \in \ell\}|.$$

How large can $I(P, L)$ be? More precisely, determine or estimate $\max_{|P|=m,|L|=n} I(P, L)$.

*Work on this chapter has been supported by a joint grant from the U.S.–Israel Binational Science Foundation, and by NSF Grants CCR-97–32101 and CCR-00–98246. Work by János Pach has also been supported by PSC-CUNY Research Award 63382–0032 and by Hungarian Science Foundation grant OTKA T-032452. Work by Micha Sharir has also been supported by a grant from the Israeli Academy of Sciences for a Center of Excellence in Geometric Computing at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

This simplest formulation of the incidence problem, due to Erdős and first settled by Szemerédi and Trotter, has been the starting point of extensive research that has picked up considerable momentum during the past two decades. It is the purpose of this survey to review the results obtained so far, describe the main techniques used in the analysis of this problem, and discuss many variations and extensions.

The problem can be generalized in many natural directions. One can ask the same question when the set L of lines is replaced by a set C of n curves of some other simple shape; the two cases involving respectively unit circles and arbitrary circles are of particular interest—see below.

A related problem involves the same kind of input—a set P of m points and a set C of n curves, but now we assume that no point of P lies on any curve of C. Let $\mathcal{A}(C)$ denote the *arrangement* of the curves of C, i.e., the decomposition of the plane into connected open cells of dimensions 0, 1, and 2 induced by drawing the elements of C. These cells are called *vertices, edges,* and *faces* of the arrangement, respectively. The total number of these cells is said to be the *combinatorial complexity* of the arrangement. See [21, 46] for details concerning arrangements. The combinatorial complexity of a single *face* is defined as the number of lower dimensional cells (i.e., vertices and edges) belonging to its boundary. The points of P then mark certain faces in the arrangement $\mathcal{A}(C)$ of the curves (assume for simplicity that there is at most one point of P in each face), and the goal is to establish an upper bound on $K(P, C)$, the combined combinatorial complexity of the marked faces. This problem is often referred to in the literature as the *Many-Faces Problem.*

One can extend the above questions to d-dimensional spaces, for $d > 2$. Here we can either continue to consider incidences between points and *curves,* or incidences between points and *surfaces* of any larger dimension k, $1 < k < d$. In the special case when $k = d-1$, we may also wish to study the natural generalization of the 'many-faces problem' described in the previous paragraph: to estimate the total combinatorial complexity of m marked (d-dimensional) cells in the arrangement of n given surfaces.

All of the above problems have many algorithmic variants. Perhaps the simplest question of this type is *Hopcroft's problem:* Given m points and n lines in the plane, how fast can one determine whether there exists any point that lies on any line? One can consider more general problems, like counting the number of incidences or reporting all of them, doing the same for a collection of curves rather than lines, computing m marked faces in an arrangement of n curves, and so on.

It turned out that two exciting *metric* problems (involving interpoint distances) proposed by Erdős in 1946 are strongly related to problems involving incidences.

1. *Repeated Distances Problem:* Given a set P of n points in the plane, what is the maximum number of pairs that are at distance exactly 1 from each other? To see the connection, let C be the set of unit circles centered at the points of P. Then two points $p, q \in P$ are at distance 1 apart if and only if the circle centered at

p passes through q and vice versa. Hence, $I(P, C)$ is twice the number of unit distances determined by P.

2. *Distinct Distances Problem:* Given a set P of n points in the plane, at least how many distinct distances must there always exist between its point pairs? Later we will show the connection between this problem and the problem of incidences between P and an appropriate set of circles of different radii.

Some other applications of the incidence problem and the many-faces problem will be reviewed at the end of this paper. They include the analysis of the maximum number of isosceles triangles, or triangles with a fixed area or perimeter, whose vertices belong to a planar point set; estimating the maximum number of mutually congruent simplices determined by a point set in higher dimensions; etc.

Historical perspective and overview. The first derivation of the tight upper bound

$$I(P, L) = \Theta(m^{2/3}n^{2/3} + m + n)$$

was given by Szemerédi and Trotter in their 1983 seminal paper [52]. They proved Erdős' conjecture, who found the matching lower bound (which was rediscovered many years later by Edelsbrunner and Welzl [25]). A slightly different lower bound construction was exhibited by Elekes [26] (see Section 2).

The original proof of Szemerédi and Trotter is rather involved, and yields a rather astronomical constant of proportionality hidden in the O-notation. A considerably simpler proof was found by Clarkson et al. [19] in 1990, using extremal graph theory combined with a geometric partitioning technique based on random sampling (see Section 3). Their paper contains many extensions and generalizations of the Szemerédi-Trotter theorem. Many further extensions can be found in subsequent papers by Edelsbrunner et al. [22, 23], by Agarwal and Aronov [1], by Aronov et al. [10], and by Pach and Sharir [41].

The next breakthrough occurred in 1997. In a surprising paper, Székely [51] gave an embarrassingly short proof (which we will review in Section 4) of the upper bound on $I(P, L)$ using a simple lower bound of Ajtai et al. [8] and of Leighton [34] on the *crossing number* of a graph G, i.e., the minimum number of edge crossings in the best drawing of G in the plane, where the edges are represented by Jordan arcs. In the literature this result is often referred to as the 'Crossing Lemma.' Székely's method can easily be extended to several other variants of the problem, but appears to be less general than the previous technique of Clarkson et al. [19].

Székely's paper has triggered an intensive re-examination of the problem. In particular, several attempts were made to improve the existing upper bound on the number of incidences between m points and n circles of arbitrary radii in the plane [42]. This was the simplest instance where Székely's proof technique failed. By combining Székely's method with a seemingly unrelated technique of Tamaki and Tokuyama [53] for cutting circles into 'pseudo-segments', Aronov and Sharir [13] managed to obtain an improved bound for this variant of the problem. Their work has then been followed by Agarwal

et al. [2], who studied the complexity of many faces in arrangements of circles and pseudo-segments, and by Agarwal et al. [5], who extended this result to arrangements of pseudo-circles (see Section 5). Aronov et al. [11] generalized the problem to higher dimensions, while Sharir and Welzl [47] studied incidences between points and *lines* in three dimensions (see Section 6).

The related problems involving distances in a point set have had mixed success in recent studies. As for the Repeated Distances Problem in the plane, the best known upper bound on the number of times the same distance can occur among n points is $O(n^{4/3})$, which was obtained nearly 20 years ago by Spencer et al. [50]. This is far from the best known lower bound of Erdős, which is slightly super-linear (see [40]). The best known upper bound for the 3-dimensional case, due to Clarkson et al. [19], is roughly $O(n^{3/2})$, while the corresponding lower bound of Erdős is $\Omega(n^{4/3} \log\log n)$ (see [39]). Many variants of the problem have been studied; see, e.g., [28].

While the Repeated Distances problem has been "stuck" for quite some time, more progress has been made on the companion problem of Distinct Distances. In the planar case, L. Moser [38], Chung [17], and Chung et al. [18] proved that the number of distinct distances determined by n points in the plane is $\Omega(n^{2/3})$, $\Omega(n^{5/7})$, and $\Omega(n^{4/5}/\text{polylog}(n))$, respectively. Székely [51] managed to get rid of the polylogarithmic factor, while Solymosi and Tóth [48] improved this bound to $\Omega(n^{6/7})$. This was a real breakthrough. Their analysis was subsequently refined by Tardos [54] and then by Katz and Tardos [33], who obtained the current record of $\Omega(n^{(48-14e)/(55-16e)-\varepsilon})$, for any $\varepsilon > 0$, which is $\Omega(n^{0.8641})$. In spite of this steady improvement, there is still a considerable gap to the best known upper bound of $O(n/\sqrt{\log n})$, due to Erdős [27] (see Section 7). In three dimensions, a recent result of Aronov et al. [12] yields a lower bound of $\Omega(n^{77/141-\varepsilon})$, for any $\varepsilon > 0$, which is $\Omega(n^{0.546})$. This is still far from the best known upper bound of $O(n^{2/3})$. A better lower bound of $\Omega(n^{0.5794})$ in a special case (involving "homogeneous" point sets) has recently been given by Solymosi and Vu [49]. Their analysis also applies to higher-dimensional homogeneous point sets, and yields the bound $\Omega(n^{2/d-1/d^2})$. In a sill unpublished manuscript, the same authors have tackled the general case, and obtained a lower bound of $\Omega(n^{2/d-1/d(d+2)})$.

For other surveys on related subjects, consult [15], Chapter 4 of [36], [39], and [40].

2. Lower Bounds

We describe a simple construction due to Elekes [26] of a set P of m points and a set L of n lines, so that $I(P, L) = \Omega(m^{2/3}n^{2/3} + m + n)$. We fix two integer parameters ξ, η. We take P to be the set of all lattice points in $\{1, 2, \ldots, \xi\} \times \{1, 2, \ldots, 2\xi\eta\}$. The set L consists of all lines of the form $y = ax + b$, where a is an integer in the range $1, \ldots, \eta$, and b is an integer in the range $1, \ldots, \xi\eta$. Clearly, each line in L passes through exactly ξ points of P. See Figure 1.

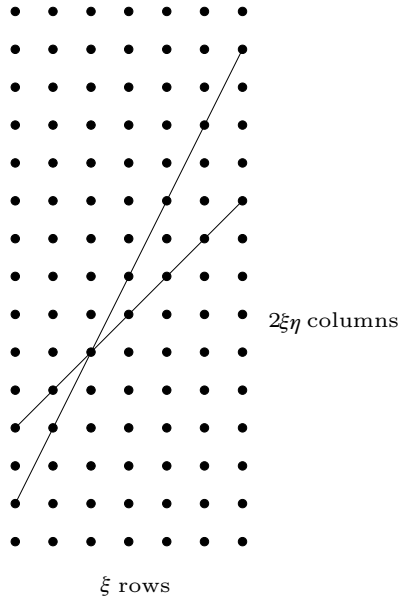


Figure 1. Elekes' construction.

We have $m = |P| = 2\xi^2\eta$, $n = |L| = \xi\eta^2$, and
$I(P, L) = \xi|L| = \xi^2\eta^2 = \Omega(m^{2/3}n^{2/3})$.

Given any sizes m, n so that $n^{1/2} \leq m \leq n^2$, we can find ξ, η that give rise to sets P, L whose sizes are within a constant factor of m and n, respectively. If m lies outside this range then $m^{2/3}n^{2/3}$ is dominated by $m + n$, and then it is trivial to construct sets P, L of respective sizes m, n so that $I(P, L) = \Omega(m + n)$. We have thus shown that

$$I(P, L) = \Omega(m^{2/3}n^{2/3} + m + n).$$

We note that this construction is easy to generalize to incidences involving other curves. For example, we can take P to be the grid $\{1, 2, \ldots, \xi\} \times \{1, 2, \ldots, 3\xi^2\eta\}$, and define C to be the set of all parabolas of the form $y = ax^2 + bx + c$, where $a \in \{1, \ldots, \eta\}$, $b \in \{1, \ldots, \xi\eta\}$, $c \in \{1, \ldots, \xi^2\eta\}$. Now we have $m = |P| = 3\xi^3\eta$, $n = |C| = \xi^3\eta^3$, and

$$I(P, C) = \xi|C| = \xi^4\eta^3 = \Omega(m^{1/2}n^{5/6}).$$

Note that in the construction we have $m = O(n)$. When m is larger, we use the preceding construction for points and lines, which can be easily transformed into a construction for points and parabolas, to obtain the overall lower bound for points and parabolas:

$$I(P, C) = \begin{cases} \Omega(m^{2/3}n^{2/3} + m), & \text{if } m \geq n \\ \Omega(m^{1/2}n^{5/6} + n), & \text{if } m \leq n. \end{cases}$$

These constructions can be generalized to incidences involving graphs of polynomials of higher degrees.

From incidences to many faces. Let P be a set of m points and L a set of n lines in the plane, and put $I = I(P, L)$. Fix a sufficiently small parameter $\varepsilon > 0$, and replace each line $\ell \in L$ by two lines ℓ^+, ℓ^-, obtained by translating ℓ parallel to itself by distance ε in the two possible directions. We obtain a new collection L' of $2n$ lines. If ε is sufficiently small then each point $p \in P$ that is incident to $k \geq 2$ lines of L becomes a point that lies in a small face of $\mathcal{A}(L')$ that has $2k$ edges; note also that the circle of radius ε centered at p is tangent to all these edges. Moreover, these faces are distinct for different points p, when ε is sufficiently small.

We have thus shown that $K(P, L') \geq 2I(P, L) - 2m$ (where the last term accounts for points that lie on just one line of L). In particular, in view of the preceding construction, we have, for $|P| = m$, $|L| = n$,

$$K(P, L) = \Omega(m^{2/3}n^{2/3} + m + n).$$

An interesting consequence of this construction is as follows. Take $m = n$ and sets P, L that satisfy $I(P, L) = \Theta(n^{4/3})$. Let C be the collection of the $2n$ lines of L' and of the n circles of radius ε centered at the points of P. By applying an inversion,[1] we can turn all the curves in C into circles. We thus obtain a set C' of $3n$ circles with $\Theta(n^{4/3})$ tangent pairs. If we replace each of the circles centered at the points of P by circles with a slightly larger radius, we obtain a collection of $3n$ circles with $\Theta(\mathrm{n}^{4/3})$ *empty lenses,* namely faces of degree 2 in their arrangement. Empty lenses play an important role in the analysis of incidences between points and circles; see Section 5.

Lower bounds for incidences with unit circles. As noted, this problem is equivalent to the problem of Repeated Distances. Erdős [27] has shown that, for the vertices of an $n^{1/2} \times n^{1/2}$ grid, there exists a distance that occurs $\Omega(n^{1+c/\log\log n})$ times, for an appropriate absolute constant $c > 0$. The details of this analysis, based on number-theoretic considerations, can be found in the monographs [36] and [40].

Lower bounds for incidences with arbitrary circles. As we will see later, we are still far from a sharp bound on the number of incidences between points and circles, especially when the number of points is small relative to the number of circles.

By taking sets P of m points and L of n lines with $I(P, L) = \Theta(m^{2/3}n^{2/3} + m + n)$, and by applying inversion to the plane, we obtain a set C of n circles and a set P' of m points with $I(P', C) = \Theta(m^{2/3}n^{2/3} + m + n)$. Hence the maximum number of incidences between m points and n circles is $\Omega(m^{2/3}n^{2/3} + m + n)$. However, we can slightly increase this lower bound, as follows.

[1] An inversion about, say, the unit circle centered at the origin, maps each point (x, y) to the point $(\frac{x}{x^2+y^2}, \frac{y}{x^2+y^2})$. It maps lines to circles passing through the origin.

Let P be the set of vertices of the $m^{1/2} \times m^{1/2}$ integer lattice. As shown by Erdős [27], there are $t = \Theta(m/\sqrt{\log m})$ distinct distances between pairs of points of P. Draw a set C of mt circles, centered at the points of P and having as radii the t possible inter-point distances. Clearly, the number of incidences $I(P, C)$ is exactly $m(m-1)$. If the bound on $I(P, C)$ were $O(m^{2/3}n^{2/3} + m + n)$, then we would have

$$m(m-1) = I(P, C) = O(m^{2/3}(mt)^{2/3} + mt) = O(m^2/((\log m)^{1/3}),$$

a contradiction. This shows that, under the most optimistic conjecture, the maximum value of $I(P, C)$ should be larger than the corresponding bound for lines by at least some polylogarithmic factor.

3. Upper Bounds for Incidences via the Partition Technique

The approach presented in this section is due to Clarkson et al. [19]. It predated Székely's method, but it seems to be more flexible, and suitable for generalizations. It can also be used for the refinement of some proofs based on Székely's method.

We exemplify this technique by establishing an upper bound for the number of point-line incidences. Let P be a set of m points and L a set of n lines in the plane. First, we give a weaker bound on $I(P, L)$, as follows. Consider the bipartite graph $H \subseteq P \times L$ whose edges represent all incident pairs (p, ℓ), for $p \in P$, $\ell \in L$. Clearly, H does not contain $K_{2,2}$ as a subgraph. By the Kövari-Sós-Turán Theorem in extremal graph theory (see [40]), we have

$$I(P, L) = O(mn^{1/2} + n). \tag{1}$$

To improve this bound, we partition the plane into subregions, apply this bound within each subregion separately, and sum up the bounds. We fix a parameter r, $1 \le r \le n$, whose value will be determined shortly, and construct a so-called $(1/r)$-*cutting* of the arrangement $\mathcal{A}(L)$ of the lines of L. This is a decomposition of the plane into $O(r^2)$ vertical trapezoids with pairwise disjoint interiors, such that each trapezoid is crossed by at most n/r lines of L. The existence of such a cutting has been established by Chazelle and Friedman [16], following earlier and somewhat weaker results of Clarkson and Shor [20]. See [36] and [46] for more details.

For each cell τ of the cutting, let P_τ denote the set of points of P that lie in the interior of τ, and let L_τ denote the set of lines that cross τ. Put $m_\tau = |P_\tau|$ and $n_\tau = |L_\tau| \le n/r$. Using (1), we have

$$I(P_\tau, L_\tau) = O\left(m_\tau n_\tau^{1/2} + n_\tau\right) = O\left(m_\tau \left(\frac{n}{r}\right)^{1/2} + \frac{n}{r}\right).$$

Summing this over all $O(r^2)$ cells τ, we obtain a total of

$$\sum_{\tau} I(P_\tau, L_\tau) = O\left(m\left(\frac{n}{r}\right)^{1/2} + nr\right)$$

incidences. This does not quite complete the count, because we also need to consider points that lie on the boundary of the cells of the cutting. A point p that lies in the relative interior of an edge e of the cutting lies on the boundary of at most two cells, and any line that passes through p, with the possible exception of the single line that contains e, crosses both cells. Hence, we may simply assign p to one of these cells, and its incidences (except for at most one) will be counted within the subproblem associated with that cell. Consider then a point p which is a vertex of the cutting, and let l be a line incident to p. Then l either crosses or bounds some adjacent cell τ. Since a line can cross the boundary of a cell in at most two points, we can charge the incidence (p, l) to the pair (l, τ), use the fact that no cell is crossed by more than n/r lines, and conclude that the number of incidences involving vertices of the cutting is at most $O(nr)$.

We have thus shown that

$$I(P, L) = O\left(m\left(\frac{n}{r}\right)^{1/2} + nr\right).$$

Choose $r = m^{2/3}/n^{1/3}$. This choice makes sense provided that $1 \le r \le n$. If $r < 1$, then $m < n^{1/2}$ and (1) implies that $I(P, L) = O(n)$. Similarly, if $r > n$ then $m > n^2$ and (1) implies that $I(P, L) = O(m)$. If r lies in the desired range, we get $I(P, L) = O(m^{2/3}n^{2/3})$. Putting all these bounds together, we obtain the bound

$$I(P, L) = O(m^{2/3}n^{2/3} + m + n),$$

as required.

Remark. An equivalent statement is that, for a set P of m points in the plane, and for any integer $k \le m$, the number of lines that contain at least k points of P is at most

$$O\left(\frac{m^2}{k^3} + \frac{m}{k}\right).$$

Discussion. The cutting-based method is quite powerful, and can be extended in various ways. The crux of the technique is to derive somehow a weaker (but easier) bound on the number of incidences, construct a $(1/r)$-cutting of the set of curves, obtain the corresponding decomposition of the problem into $O(r^2)$ subproblems, apply the weaker bound within each subproblem, and sum up the bounds to obtain the overall bound. The work by Clarkson et al. [19] contains many such extensions.

Let us demonstrate this method to obtain an upper bound for the number of incidences between a set P of m points and a set C of n arbitrary circles in the plane.

Here the forbidden subgraph property is that the incidence graph $H \subseteq P \times C$ does not contain $K_{3,2}$ as a subgraph, and thus (see [40])

$$I(P, C) = O(mn^{2/3} + n).$$

We construct a $(1/r)$-cutting for C, apply this weak bound within each cell τ of the cutting, and handle incidences that occur on the cell boundaries exactly as above, to obtain

$$I(P, C) = \sum_{\tau} I(P_\tau, C_\tau) = O\left(m\left(\frac{n}{r}\right)^{2/3} + nr\right).$$

With an appropriate choice of $r = m^{3/5}/n^{1/5}$, this becomes

$$I(P, C) = O(m^{3/5}n^{4/5} + m + n).$$

However, as we shall see later, in Section 5, this bound can be considerably improved.

The case of a set C of n unit circles is handled similarly, observing that in this case the intersection graph H does not contain $K_{2,3}$. This yields the same upper bound $I(P, C) = O(mn^{1/2} + n)$, as in (1). The analysis then continues exactly as in the case of lines, and yields the bound

$$I(P, C) = O(m^{2/3}n^{2/3} + m + n).$$

We can apply this bound to the Repeated Distances Problem, recalling that the number of pairs of points in an n-element set of points in the plane that lie at distance exactly 1 from each other, is half the number of incidences between the points and the unit circles centered at them. Substituting $m = n$ in the above bound, we thus obtain that the number of repeated distances is at most $O(n^{4/3})$. This bound is far from the best known lower bound, mentioned in Section 2, and no improvement has been obtained since its original derivation in [50] in 1984.

As a matter of fact, this approach can be extended to any collection C of curves that have "d degrees of freedom", in the sense that any d points in the plane determine at most $t = O(1)$ curves from the family that pass through all of them, and any pair of curves intersect in only $O(1)$ points. The incidence graph does not contain $K_{d,t+1}$ as a subgraph, which implies that

$$I(P, C) = O(mn^{1-1/d} + n).$$

Combining this bound with a cutting-based decomposition yields the bound

$$I(P, C) = O(m^{d/(2d-1)}n^{(2d-2)/(2d-1)} + m + n).$$

Note that this bound extrapolates the previous bounds for the cases of lines ($d = 2$), unit circles ($d = 2$), and arbitrary circles ($d = 3$). See [42] for a slight generalization of this result, using Székely's method, outlined in the following section.

4. Incidences via Crossing Numbers—Székely's Method

A graph G is said to be *drawn* in the plane if its vertices are mapped to distinct points in the plane, and each of its edges is represented by a Jordan are connecting the corresponding pair of points. It is assumed that no edge passes through any vertex other than its endpoints, and that when two edges meet at a common interior point, they properly *cross* each other there, i.e., each curve passes from one side of the other curve to the other side. Such a point is called a *crossing*. In the literature, a graph drawn in the plane with the above properties is often called a *topological graph*. If, in addition, the edges are represented by straight-line segments, then the drawing is said to be a *geometric graph*.

As we have indicated before, Székely discovered that the analysis outlined in the previous section can be substantially simplified, applying the following so-called Crossing Lemma for graphs drawn in the plane.

Lemma 4.1 (Leighton [34], Ajtai et al. [8]) *Let G be a simple graph drawn in the plane with V vertices and E edges. If $E > 4V$ then there are $\Omega(E^3/V^2)$ crossing pairs of edges.*

To establish the lemma, denote by $\mathrm{cr}(G)$ the minimum number of crossing pairs of edges in any 'legal' drawing of G. Since G contains too many edges, it is not planar, and therefore $\mathrm{cr}(G) \geq 1$. In fact, using Euler's formula, a simple counting argument shows that $\mathrm{cr(G)} \geq E - 3V + 6 > E - 3V$. We next apply this inequality to a random sample G' of G, which is an induced subgraph obtained by choosing each vertex of G independently with some probability p. By applying expectations, we obtain $\mathbf{E}[\mathrm{cr}(G')] \geq \mathbf{E}[E'] - 3\mathbf{E}[V']$, where E', V' are the numbers of edges and vertices in G', respectively. This can be rewritten as $\mathrm{cr}(G)p^4 \geq Ep^2 - 3Vp$, and choosing $p = 4V/E$ completes the proof of Lemma 4.1.

We remark that the constant of proportionality in the asserted bound, as yielded by the preceding proof, is 1/64, but it has been improved by Pach and Tóth [44]. They proved that $\mathrm{cr}(G) \geq E^3/(33.75\mathrm{V}^2)$ whenever $E \geq 7.5V$. In fact, the slightly weaker inequality $\mathrm{cr}(G) \geq E^3/(33.75V^2) - 0.9V$ holds without any extra assumption. We also note that it is crucial that the graph G be *simple* (i.e., any two vertices be connected by at most one edge), for otherwise no crossing can be guaranteed, regardless of how large E is.

Let P be a set of m points and L a set of n lines in the plane. We associate with P and L the following plane drawing of a graph G. The vertices of (this drawing of) G are the points of P. For each line $\ell \in L$, we connect each pair of points of $P \cap \ell$ that are

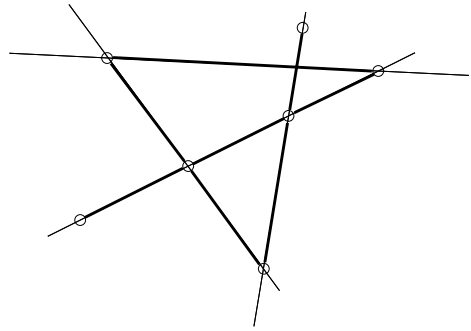


Figure 2. Székely's graph for points and lines in the plane.

consecutive along ℓ by an edge of G, drawn as the straight segment between these points (which is contained in ℓ). See Figure 2 for an illustration. Clearly, G is a simple graph, and, assuming that each line of L contains at least one point of P, we have $V = m$ and $E = I(P, L) - n$ (the number of edges along a line is smaller by 1 than the number of incidences with that line). Hence, either $E < 4V$, and then $I(P, L) < 4m + n$, or $\mathrm{cr}(G) \geq E^3/(cV^2) = (I(P, L) - n)^3/(cm^2)$. However, we have, trivially, $\mathrm{cr}(G) \leq \binom{n}{2}$, implying that $I(P, L) \leq (c/2)^{1/3}m^{2/3}n^{2/3} + n \leq 2.57m^{2/3}n^{2/3} + n$.

Extensions: Many faces and unit circles. The simple idea behind Székely's proof is quite powerful, and can be applied to many variants of the problem, as long as the corresponding graph G is simple, or, alternatively, has a bounded edge multiplicity. For example, consider the case of incidences between a set P of m points and a set C of n unit circles. Draw the graph G exactly as in the case of lines, but only along circles that contain more than two points of P, to avoid loops and multiple edges along the same circle. We have $V = m$ and $E \geq I(P, C) - 2n$. In this case, G need not be simple, but the maximum edge multiplicity is at most two; see Figure 3. Hence, by

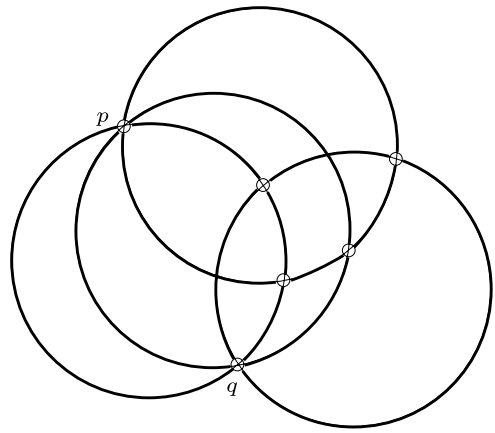


Figure 3. Székely's graph for points and unit circles in the plane: The maximum edge multiplicity is two—see the edges connecting p and q.

deleting at most half of the edges of G we make it into a simple graph. Moreover, $\mathrm{cr}(G) \leq n(n-1)$, so we get $I(P, C) = O(m^{2/3}n^{2/3} + m + n)$, again with a rather small constant of proportionality.

We can also apply this technique to obtain an upper bound on the complexity of many faces in an arrangement of lines. Let P be a set of m points and L a set of n lines in the plane, so that no point lies on any line and each point lies in a distinct face of $\mathcal{A}(L)$. The graph G is now constructed in the following slightly different manner. Its vertices are the points of P. For each $\ell \in L$, we consider all faces of $\mathcal{A}(L)$ that are marked by points of P, are bounded by ℓ and lie on a fixed side of ℓ. For each pair f_1, f_2 of such faces that are consecutive along ℓ (the portion of ℓ between ∂f_1 and ∂f_2 does not meet any other marked face on the same side), we connect the corresponding marking points p_1, p_2 by an edge, and draw it as a polygonal path $p_1q_1q_2p_2$, where $q_1 \in \ell \cap \partial f_1$ and $q_2 \in \ell \cap \partial f_2$. We actually shift the edge slightly away from ℓ so as to avoid its overlapping with edges drawn for faces on the other side of ℓ. The points q_1, q_2 can be chosen in such a way that a pair of edges meet each other only at intersection points of pairs of lines of L. See Figure 4. Here we have $V = m$, $E \geq K(P, L) - 2n$, and $\mathrm{cr}(G) \leq 2n(n-1)$ (each pair of lines can give rise to at most four pairs of crossing edges, near the same intersection point). Again, G is not simple, but the maximum edge multiplicity is at most two, because, if two faces f_1, f_2 are connected along a line ℓ, then ℓ is a common external tangent to both faces. Since f_1 and f_2 are disjoint convex sets, they can have at most two external common tangents. Hence, arguing as above, we obtain $K(P, L) = O(m^{2/3}n^{2/3} + m + n)$. We remark that the same upper bound can also be obtained via the partition technique, as shown by Clarkson et al. [19]. Moreover, in view of the discussion in Section 2, this bound is tight.

However, Székely's technique does not always apply. The simplest example where it fails is when we want to establish an upper bound on the number of incidences

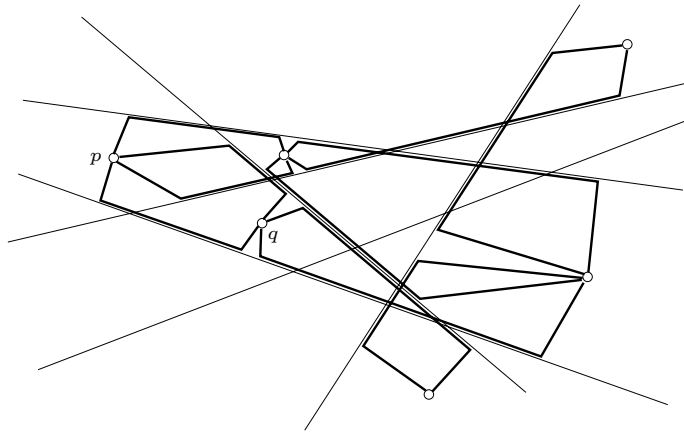


Figure 4. Székely's graph for face-marking points and lines in the plane. The maximum edge multiplicity is two—see, e.g., the edges connecting p and q.

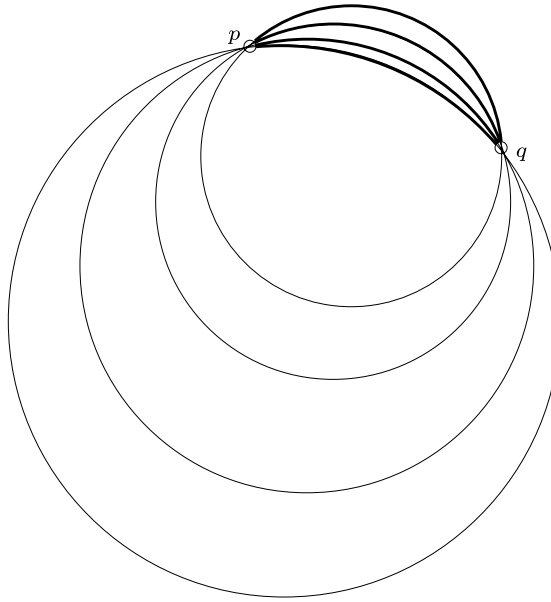


Figure 5. Székely's graph need not be simple for points and arbitrary circles in the plane.

between points and circles of arbitrary radii. If we follow the same approach as for equal circles, and construct a graph analogously, we may now create edges with arbitrarily large multiplicities, as is illustrated in Figure 5. We will tackle this problem in the next section.

Another case where the technique fails is when we wish to bound the total complexity of many faces in an arrangement of line *segments*. If we try to construct the graph in the same way as we did for full lines, the faces may not be convex any more, and we can create edges of high multiplicity; see Figure 6.

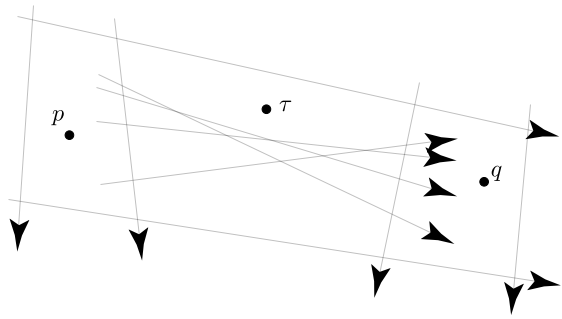


Figure 6. Székely's graph need not be simple for marked faces and segments in the plane: An arbitrarily large number of segments bounds all three faces marked by the points p, q, r, so the edges (p, r) and (r, q) in Székely's graph have arbitrarily large multiplicity.

5. Improvements by Cutting into Pseudo-segments

Consider the case of incidences between points and circles of arbitrary radii. One way to overcome the technical problem in applying Székely's technique in this case is to cut the given circles into arcs so that any two of them intersect at most once. We refer to such a collection of arcs as a collection of *pseudo-segments.*

The first step in this direction has been taken by Tamaki and Tokuyama [53], who have shown that any collection C of n *pseudo-circles,* namely, closed Jordan curves, each pair of which intersect at most twice, can be cut into $O(n^{5/3})$ arcs that form a family of pseudosegments. The union of two arcs that belong to distinct pseudo-circles and connect the same pair of points is called a *lens.* Let $\chi(C)$ denote the minimum number of points that can be removed from the curves of C, so that any two members of the resulting family of arcs have at most one point in common. Clearly, every lens must contain at least one of these cutting points, so Tamaki and Tokuyama's problem asks in fact for an upper bound on the number of points needed to "stab" all lenses. Equivalently, this problem can be reformulated, as follows.

Consider a hypergraph H whose vertex set consists of the edges of the arrangement $\mathcal{A}(C)$, i.e., the arcs between two consecutive crossings. Assign to each lens a *hyperedge* consisting of all arcs that belong to the lens. We are interested in finding the *transversal number* (or the size of the smallest "hitting set") of H, i.e., the smallest number of vertices of H that can be picked with the property that every hyperedge contains at least one of them. Based on Lovász' analysis [35] (see also [40]) of the greedy algorithm for bounding the transversal number from above (i.e., for constructing a hitting set), this quantity is not much bigger than the size of the largest *matching* in H, i.e., the maximum number of pairwise disjoint hyperedges. This is the same as the largest number of pairwise non-overlapping lenses, that is, the largest number of lenses, no two of which share a common edge of the arrangement $\mathcal{A}(C)$ (see Figure 7). Viewing

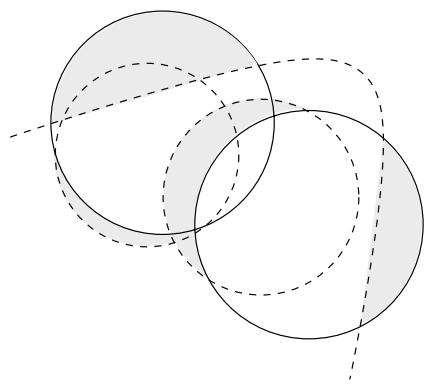


Figure 7. The boundaries of the shaded regions are nonoverlapping lenses in an arrangement of pseudo-circles. (Observe that the *regions* bounded by nonoverlapping lenses can overlap, as is illustrated here.)

such a family as a graph G, whose edges connect pairs of curves that form a lens in the family, Tamaki and Tokuyama proved that G does not contain $K_{3,3}$ as a subgraph, and this leads to the asserted bound on the number of cuts.

In order to establish an upper bound on the number of incidences between a set of m points P and a set of n circles (or pseudo-circles) C, let us construct a modified version G' of Székely's graph: its vertices are the points of P, and its edges connect adjacent pairs of points along the new pseudo-segment arcs. That is, we do not connect a pair of points that are adjacent along an original curve, if the arc that connects them has been cut by some point of the hitting set. Moreover, as in the original analysis of Székely, we do not connect points along pseudo-circles that are incident to only one or two points of P, to avoid loops and trivial multiplicities.

Clearly, the graph G' is simple, and the number E' of its edges is at least $I(P, C) - \chi(C) - 2n$. The crossing number of G' is, as before, at most the number of crossings between the original curves in C, which is at most $n(n-1)$. Using the Crossing Lemma (Lemma 4.1), we thus obtain

$$I(P, C) = O(m^{2/3}n^{2/3} + \chi(C) + m + n).$$

Hence, applying the Tamaki-Tokuyama bound on $\chi(C)$, we can conclude that

$$I(P, C) = O(m^{2/3}n^{2/3} + n^{5/3} + m).$$

An interesting property of this bound is that it is tight when $m \geq n^{3/2}$. In this case, the bound becomes $I(P, C) = O(m^{2/3}n^{2/3} + m)$, matching the lower bound for incidences between points and lines, which also serves as a lower bound for the number of incidences between points and circles or parabolas. However, for smaller values of m, the term $O(n^{5/3})$ dominates, and the dependence on m disappears. This can be rectified by combining this bound with a cutting-based problem decomposition, similar to the one used in the preceding section, and we shall do so shortly.

Before proceeding, though, we note that Tamaki and Tokuyama's bound is not tight. The best known lower bound is $\Omega(n^{4/3})$, which follows from the lower bound construction for incidences between points and lines. (That is, we have already seen that this construction can be modified so as to yield a collection C of n circles with $\Theta(n^{4/3})$ empty lenses. Clearly, each such lens requires a separate cut, so $\chi(C) = \Omega(n^{4/3})$.) Recent work by Alon et al. [9], Aronov and Sharir [13], and Agarwal et al. [5] has led to improved bounds. Specifically, it was shown in [5] that $\chi(C) = O(n^{8/5})$, for families C of *pseudo-parabolas* (graphs of continuous everywhere defined functions, each pair of which intersect at most twice), and, more generally, for families of *x-monotone* pseudo-circles (closed Jordan curves with the same property, so that the two portions of their boundaries connecting their leftmost and rightmost points are graphs of two continuous functions, defined on a common interval).

In certain special cases, including the cases of circles and of vertical parabolas (i.e., parabolas of the form $y = ax^2 + bx + c$), one can do better, and show that

$$\chi(C) = O(n^{3/2}k(n)),$$

where

$$\kappa(n) = (\log n)^{O(\alpha^2(n))},$$

and where $\alpha(n)$ is the extremely slowly growing inverse Ackermann's function. This bound was established in [5], and it improves a slightly weaker bound obtained by Aronov et al. [13]. The technique used for deriving this result is interesting in its own right, and raises several deep open problems, which we omit in this survey.

With the aid of this improved bound on $\chi(C)$, the modification of Székely's method reviewed above yields, for a set C of n circles and a set P of m points,

$$I(P, C) = O(m^{2/3}n^{2/3} + n^{3/2}\kappa(n) + m).$$

As already noted, this bound is tight when it is dominated by the first or last terms, which happens when m is roughly larger than $n^{5/4}$. For smaller values of m, we decompose the problem into subproblems, using the following so-called "dual" partitioning technique. We map each circle $(x - a)^2 + (y - b)^2 = \rho^2$ in C to the "dual" point $(a, b, \rho^2 - a^2 - b^2)$ in 3-space, and map each point (ξ, η) of P to the "dual" plane $z = -2\xi x - 2\eta y + (\xi^2 + \eta^2)$. As is easily verified, each incidence between a point of P and a circle of C is mapped to an incidence between the dual plane and point. We now fix a parameter r, and construct a $(1/r)$-cutting of the arrangement of the dual planes, which partitions $\mathbb{R}^3$ into $O(r^3)$ cells (which is a tight bound in the case of planes), each crossed by at most m/r dual planes and containing at most n/r^3 dual points (the latter property, which is not an intrinsic property of the cutting, can be enforced by further partitioning cells that contain more than n/r^3 points). We apply, for each cell τ of the cutting, the preceding bound for the set P_τ of points of P whose dual planes cross τ, and for the set C_τ of circles whose dual points lie in τ. (Some special handling of circles whose dual points lie on boundaries of cells of the cutting is needed, as in Section 3, but we omit the treatment of this special case.) This yields the bound

$$\begin{aligned} I(P, C) &= O(r^3) \cdot O\left(\left(\frac{m}{r}\right)^{2/3}\left(\frac{n}{r^3}\right)^{2/3} + \left(\frac{n}{r^3}\right)^{3/2}\kappa\left(\frac{n}{r^3}\right) + \frac{m}{r}\right) \\ &= O\left(m^{2/3}n^{2/3}r^{1/3} + \frac{n^{3/2}}{r^{3/2}}\kappa\left(\frac{n}{r^3}\right) + mr^2\right). \end{aligned}$$

Assume that m lies between $n^{1/3}$ and $n^{5/4}$, and choose $r = n^{5/11}/m^{4/11}$ in the last bound, to obtain

$$I(P, C) = O(m^{2/3}n^{2/3} + m^{6/11}n^{9/11}\kappa(m^3/n) + m + n).$$

It is not hard to see that this bound also holds for the complementary ranges of m.

6. Incidences in Higher Dimensions

It is natural to extend the study of incidences to instances involving points and curves or surfaces in higher dimensions. The case of incidences between points and (hyper)surfaces (mainly hyperplanes) has been studied earlier. Edelsbrunner et al. [23] considered incidences between points and planes in three dimensions. It is important to note that, without imposing some restrictions either on the set P of points or on the set H of planes, one can easily obtain $|P| \cdot |H|$ incidences, simply by placing all the points of P on a line, and making all the planes of H pass through that line. Some natural restrictions are to require that no three points be collinear, or that no three planes be collinear, or that the points be vertices of the arrangement $\mathcal{A}(H)$, and so on. Different assumptions lead to different bounds. For example, Agarwal and Aronov [1] proved an asymptotically tight bound $\Theta(m^{2/3}n^{d/3} + n^{d-1})$ for the number of incidences between n hyperplanes in d dimensions and $m > n^{d-2}$ vertices of their arrangement (see also [23]), as well as for the number of facets bounding m distinct cells in such an arrangement. Edelsbrunner and Sharir [24] considered the problem of incidences between points and hyperplanes in four dimensions, under the assumption that all points lie on the upper envelope of the hyperplanes. They obtained the bound $O(m^{2/3}n^{2/3} + m + n)$ for the number of such incidences, and applied the result to obtain the same upper bound on the number of bichromatic minimal distance pairs between a set of m blue points and a set of n red points in three dimensions. Another set of bounds and related results are obtained by Brass and Knauer [14], for incidences between m points and n planes in 3-space, and also for incidences in higher dimensions.

The case of incidences between points and *curves* in higher dimensions has been studied only recently. There are only two papers that address this problem. One of them, by Sharir and Welzl [47], studies incidences between points and lines in 3-space. The other, by Aronov et al. [11], is concerned with incidences between points and circles in higher dimensions. Both works were motivated by problems asked by Elekes. We briefly review these results in the following two subsections.

6.1. Points and Lines in Three Dimensions

Let P be a set of m points and L a set of n lines in 3-space. Without making some assumptions on P and L, the problem is trivial, for the following reason. Project P and L onto some generic plane. Incidences between points of P and lines of L are bijectively mapped to incidences between the projected points and lines, so we have $I(P, L) = O(m^{2/3}n^{2/3} + m + n)$. Moreover, this bound is tight, as is shown by the planar lower bound construction. (As a matter of fact, this reduction holds in any dimension $d \geq 3$.)

There are several ways in which the problem can be made interesting. First, suppose that the points of P are *joints* in the arrangement $\mathcal{A}(L)$, namely, each point is incident to at least three non-coplanar lines of L. In this case, one has $I(P, L) = O(n^{5/3})$ [47]. Note that this bound is independent of m. In fact, it is known that the number of joints

is at most $O(n^{23/14}\log^{31/14} n)$, which is $O(n^{1.643})$ [45] (the best lower bound, based on lines forming a cube grid, is only $\Omega(n^{3/2})$).

For general point sets P, one can use a new measure of incidences, which aims to ignore incidences between a point and many incident coplanar lines. Specifically, we define the *plane cover* $\pi_L(p)$ of a point p to be the minimum number of planes that pass through p so that their union contains all lines of L incident to p, and define $I_c(P, L) = \sum_{p\in P} \pi_L(p)$. It is shown in [47] that

$$I_c(P, L) = O(m^{4/7}m^{5/7} + m + n),$$

which is smaller than the planar bound of Szemerédi and Trotter.

Another way in which we can make the problem "truly 3-dimensional" is to require that all lines in L be *equally inclined,* meaning that each of them forms a fixed angle (say, 45°) with the z-direction. In this case, every point of P that is incident to at least three lines of L is a joint, but this special case admits better upper bounds. Specifically, we have

$$I(P, L) = O(\min\left\{m^{3/4}n^{1/2}\kappa(m), m^{4/7}n^{5/7}\right\} + m + n).$$

The best known lower bound is

$$I(P, L) = \Omega(m^{2/3}n^{1/2}).$$

Let us briefly sketch the proof of the upper bound $O(m^{3/4}n^{1/2}\kappa(m))$. For each $p \in P$ let C_p denote the (double) cone whose apex is p, whose symmetry axis is the vertical line through p, and whose opening angle is 45°. Fix some generic horizontal plane π_0, and map each $p \in P$ to the circle $C_p \cap \pi_0$. Each line $\ell \in L$ is mapped to the point $\ell \cap \pi_0$, coupled with the projection ℓ^* of ℓ onto π_0. Note that an incidence between a point $p \in P$ and a line $\ell \in L$ is mapped to the configuration in which the circle dual to p is incident to the point dual to ℓ and the projection of ℓ passes through the center of the circle; see Figure 8. Hence, if a line ℓ is incident to several points $p_1, \ldots, p_k \in P$, then the dual circles $p_1^*, \ldots, p_k^*$ are all tangent to each other at the common point $\ell \cap \pi_0$. Viewing these tangencies as a collection of degenerate lenses, we can bound the overall number of these tangencies, which is equal to $I(P, L)$, by $O(n^{3/2}\kappa(n))$. By a slightly more careful analysis, again based on cutting, one can obtain the bound $O(m^{3/4}n^{1/2}\kappa(m))$.

6.2. Points and Circles in Three and Higher Dimensions

Let C be a set of n circles and P a set of m points in 3-space. Unlike in the case of lines, there is no obvious reduction of the problem to a planar one, because the projection of C onto some generic plane yields a collection of ellipses, rather than circles, which can cross each other at four points per pair. However, using a more refined analysis, Aronov et al. [11] have obtained the same asymptotic bound of

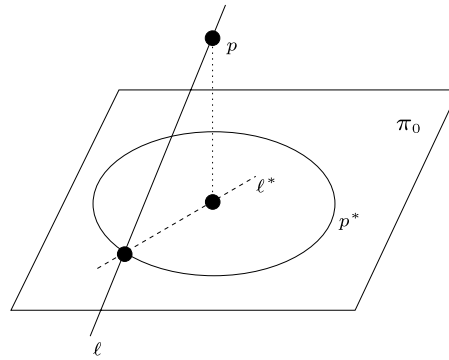


Figure 8. Transforming incidences between points and equally inclined lines to tangencies between circles in the plane.

$I(P, C) = O(m^{2/3}n^{2/3} + m^{6/11}n^{9/11}\kappa(m^3/n) + m + n)$ for $I(P, C)$. The same bound applies in any dimension $d \geq 3$.

7. Applications

The problem of bounding the number of incidences between various geometric objects is elegant and fascinating, and it has been mostly studied for its own sake. However, it is closely related to a variety of questions in combinatorial and computational geometry. In this section, we briefly review some of these connections and applications.

7.1. Algorithmic Issues

There are two types of algorithmic problems related to incidences. The first group includes problems where we wish to actually determine the number of incidences between certain objects, e.g., between given sets of points and curves, or we wish to compute (describe) a collection of marked faces in an arrangement of curves or surfaces. The second group contains completely different questions whose solution requires tools and techniques developed for the analysis of incidence problems.

In the simplest problem of the first kind, known as Hopcroft's problem, we are given a set P of m points and a set L of n lines in the plane, and we ask whether there exists at least one incidence between P and L. The best running time known for this problem is $O(m^{2/3}n^{2/3} \cdot 2^{O(\log^*(m+n))})$ [37] (see [31] for a matching lower bound). Similar running time bounds hold for the problems of counting or reporting all the incidences in $I(P, L)$. The solutions are based on constructing cuttings of an appropriate size and thereby obtaining a decomposition of the problem into subproblems, each of which can be solved by a more brute-force approach. In other words, the solution can be viewed as an implementation of the cutting-based analysis of the combinatorial bound for $I(P, L)$, as presented in Section 3.

The case of incidences between a set P of m points and a set C of n circles in the plane is more interesting, because the analysis that leads to the current best upper bound on $I(P, C)$ is not easy to implement. In particular, suppose that we have already cut the circles of C into roughly $O(n^{3/2})$ pseudo-segments (an interesting and non-trivial algorithmic task in itself), and we now wish to compute the incidences between these pseudo-segments and the points of P. Székely's technique is non-algorithmic, so instead we would like to apply the cutting-based approach to these pseudo-segments and points. However, this approach, for the case of lines, after decomposing the problem into subproblems, proceeds by duality. Specifically, it maps the points in a subproblem to dual lines, constructs the arrangement of these dual lines, and locates in the arrangement the points dual to the lines in the subproblem. When dealing with the case of pseudo-segments, there is no obvious incidence-preserving duality that maps them to points and maps the points to pseudo-lines. Nevertheless, such a duality has been recently defined by Agarwal and Sharir [7] (refining an older and less efficient duality given by Goodman [32]), which can be implemented efficiently and thus yields an efficient algorithm for computing $I(P, C)$, whose running time is comparable with the bound on $I(P, C)$ given above. A similar approach can be used to compute many faces in arrangements of pseudo-circles; see [2] and [7]. Algorithmic aspects of incidence problems have also been studied in higher dimensions; see, e.g., Brass and Knauer [14].

The cutting-based approach has by now become a standard tool in the design of efficient geometric algorithms in a variety of applications in range searching, geometric optimization, ray shooting, and many others. It is beyond the scope of this survey to discuss these applications, and the reader is referred, e.g., to the survey of Agarwal and Erickson [3] and to the references therein.

7.2. Distinct Distances

The above techniques can be applied to obtain some nontrivial results concerning the Distinct Distances problem of Erdős [27] formulated in the Introduction: what is the minimum number of distinct distances determined by n points in the plane? As we have indicated after presenting the proof of the Crossing Lemma (Lemma 4.1), Székely's idea can also be applied in several situations where the underlying graph is not *simple,* i.e., two vertices can be connected by more than one edge. However, for the method to work it is important to have an upper bound for the multiplicity of the edges. Székely [51] formulated the following natural generalization of Lemma 4.1.

Lemma. *Let G be a multigraph drawn in the plane with V vertices, E edges, and with maximal edge-multiplicity M. Then there are $\Omega\left(\frac{E^3}{MV^2}\right) - O(M^2V)$ crossing pairs of edges.*

Székely applied this statement to the Distinct Distances problem, and improved by a polylogarithmic factor the best previously known lower bound of Chung et al. [18] on

the minimum number of distinct distances determined by n points in the plane. His new bound was $\Omega(n^{4/5})$. However, Solymosi and Tóth [48] have realized that, combining Székely's analysis of distinct distances with the Szemerédi-Trotter theorem for the number of incidences between m points and n lines in the plane, this lower bound can be substantially improved. They managed to raise the bound to $\Omega(n^{6/7})$. Later, Tardos and Katz have further improved this result, using the same general approach, but improving upon a key algebraic step of the analysis. In their latest paper [33], they combined their methods to prove that the minimum number of distinct distances determined by n points in the plane is $\Omega(n^{(48-14e)/(55-16e)-\varepsilon})$, for any $\varepsilon > 0$, which is $\Omega(n^{0.8641})$. This is the best known result so far. A close inspection of the general Solymosi-Tóth approach shows that, without any additional geometric idea, it can never lead to a lower bound better than $\Omega(n^{8/9})$.

7.3. Equal-area, Equal-perimeter, and Isoceles Triangles

Let P be a set of n points in the plane. We wish to bound the number of triangles spanned by the points of P that have a *given area,* say 1. To do so, we note that if we fix two points $a, b \in P$, any third point $p \in P$ for which Area $(\Delta abp) = 1$ lies on the union of two fixed lines parallel to ab. Pairs (a, b) for which such a line ℓ_{ab} contains fewer than $n^{1/3}$ points of P generate at most $O(n^{7/3})$ unit area triangles. For the other pairs, we observe that the number of lines containing more than $n^{1/3}$ points of P is at most $O(n^2/(n^{1/3})^3) = O(n)$, which, as already mentioned, is an immediate consequence of the Szemerédi-Trotter theorem. The number of incidences between these lines and the points of P is at most $O(n^{4/3})$. We next observe that any line ℓ can be equal to one of the two lines ℓ_{ab} for at most n pairs a, b, because, given ℓ and a, there can be at most two points b for which $\ell = \ell_{ab}$. It follows that the lines containing more than $n^{1/3}$ points of P can be associated with at most $O(n \cdot n^{4/3}) = O(n^{7/3})$ unit area triangles. Hence, overall, P determines at most $O(n^{7/3})$ unit area triangles. The best known lower bound is $\Omega(n^2 \log\log n)$ (see [15]).

Next, consider the problem of estimating the number of *unit perimeter* triangles determined by P. Here we note that if we fix $a, b \in P$, with $|ab| < 1$, any third point $p \in P$ for which Perimeter$(\Delta abp) = 1$ lies on an ellipse whose foci are a and b and whose major axis is $1 - |ab|$. Clearly, any two distinct pairs of points of P give rise to distinct ellipses, and the number of unit perimeter triangles determined by P is equal to one third of the number of incidences between these $O(n^2)$ ellipses and the points of P. The set of these ellipses has four degrees of freedom, in the sense of Pach and Sharir [42] (see also Section 3), and hence the number of incidences between them and the points of P, and consequently the number of unit perimeter triangles determined by P, is at most

$$O(n^{4/7}(n^2)^{6/7}) = O(n^{16/7}).$$

Here the best known lower bound is very weak—only $\Omega(ne^{c\frac{\log n}{\log\log n}})$ [15].

Finally, consider the problem of estimating the number of *isosceles* triangles determined by P. Recently, Pach and Tardos [43] proved that the number of isosceles triangles induced by triples of an n-element point set in the plane is $O(n^{(11-3\alpha)/(5-\alpha)})$ (where the constant of proportionality depends on α), provided that $0 < \alpha < \frac{10-3e}{24-7e}$. In particular, the number of isoceles triangles is $O(n^{2.136})$. The best known lower bound is $\Omega(n^2 \log n)$ [15]. The proof proceeds through two steps, interesting in their own right.

(i) Let P be a set of n distinct points and let C be a set of ℓ distinct circles in the plane, with $m \leq \ell$ distinct centers. Then, for any $0 < \alpha < 1/e$, the number I of incidences between the points in P and the circles of C is

$$O\left(n + \ell + n^{\frac{2}{3}}\ell^{\frac{2}{3}} + n^{\frac{4}{7}}m^{\frac{1+\alpha}{7}}\ell^{\frac{5-\alpha}{7}} + n^{\frac{12+14\alpha}{21+3\alpha}}m^{\frac{3+5\alpha}{21+3\alpha}}\ell^{\frac{15-3\alpha}{21+3\alpha}} + n^{\frac{8+2\alpha}{14+\alpha}}m^{\frac{2+2\alpha}{14+\alpha}}\ell^{\frac{10-2\alpha}{14+\alpha}}\right),$$

where the constant of proportionality depends on α.

(ii) As a corollary, we obtain the following statement. Let P be a set of n distinct points and let C be a set of ℓ distinct circles in the plane such that they have at most n distinct centers. Then, for any $0 < \alpha < 1/e$, the number of incidences between the points in P and the circles in C is

$$O\left(n^{\frac{5+3\alpha}{7+\alpha}}\ell^{\frac{5-\alpha}{7+\alpha}} + n\right).$$

In view of a recent result of Katz and Tardos [33], both statements extend to all $0 < \alpha < \frac{10-3e}{24-7e}$, which easily implies the above bound on the number of isosceles triangles.

7.4. Congruent Simplices

Bounding the number of incidences between points and circles in higher dimensions can be applied to the following interesting question asked by Erdős and Purdy [29, 30] and discussed by Agarwal and Sharir [6]. Determine the largest number of simplices congruent to a fixed simplex σ, which can be spanned by an n-element point set $P \subset \mathbb{R}^k$?

Here we consider only the case when $P \subset \mathbb{R}^4$ and $\sigma = abcd$ is a 3-simplex. Fix three points $p, q, r \in P$ such that the triangle pqr is congruent to the face abc of σ. Then any fourth point $v \in P$ for which $pqrv$ is congruent to σ must lie on a circle whose plane is orthogonal to the triangle pqr, whose radius is equal to the height of σ from d, and whose center is at the foot of that height. Hence, bounding the number of congruent simplices can be reduced to the problem of bounding the number of incidences between circles and points in 4-space. (The actual reduction is slightly more involved, because the same circle can arise for more than one triangle pqr; see [6] for details.) Using the bound of [11], mentioned in Section 6, one can deduce that the number of congruent 3-simplices determined by n points in 4-space is $O(n^{20/9+\varepsilon})$, for any $\varepsilon > 0$.

This is just one instance of a collection of bounds obtained in [6] for the number of congruent copies of a k-simplex in an n-element point set in $\mathbb{R}^d$, whose review is beyond the scope of this survey.

References

[1] P.K. Agarwal and B. Aronov, Counting facets and incidences, *Discrete Comput. Geom.* 7: 359–369 (1992).

[2] P.K. Agarwal, B. Aronov and M. Sharir, On the complexity of many faces in arrangements of pseudo-segments and of circles, in *Discrete and Computational Geometry — The Goodman-Pollack Festschrift,* B. Aronov, S. Basu, J. Pach and M. Sharir (Eds.), Springer-Verlag, Heidelberg, (2003) pp. 1–23.

[3] P.K. Agarwal and J. Erickson, Geometric range searching and its relatives, in: *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman and R. Pollack, eds.), AMS Press, Providence, RI, (1998) pp. 1–56.

[4] P.K. Agarwal, A. Efrat and M. Sharir, Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications, *SIAM J. Comput.* 29: 912–953 (2000).

[5] P.K. Agarwal, E. Nevo, J. Pach, R. Pinchasi, M. Sharir and S. Smorodinsky, Lenses in arrangements of pseudocircles and their applications, *J. ACM* 51: 139–186 (2004).

[6] P.K. Agarwal and M. Sharir, On the number of congruent simplices in a point set, *Discrete Comput. Geom.* 28: 123–150 (2002).

[7] P.K. Agarwal and M. Sharir, Pseudoline arrangements: Duality, algorithms and applications, *SIAM J. Comput.*

[8] M. Ajtai, V. Chvátal, M. Newborn and E. Szemerédi, Crossing-free subgraphs, *Ann. Discrete Math* 12: 9–12 (1982).

[9] N. Alon, H. Last, R. Pinchasi and M. Sharir, On the complexity of arrangements of circles in the plane, *Discrete Comput. Geom.* 26: 465–492 (2001).

[10] B. Aronov, H. Edelsbrunner, L. Guibas and M. Sharir, Improved bounds on the number of edges of many faces in arrangements of line segments, *Combinatorica* 12: 261–274 (1992).

[11] B. Aronov, V. Koltun and M. Sharir, Incidences between points and circles in three and higher dimensions, *Discrete Comput. Geom.*

[12] B. Aronov, J. Pach, M. Sharir and G. Tardos, Distinct distances in three and higher dimensions, *Combinatorics, Probability and Computing* 13: 283–293 (2004).

[13] B. Aronov and M. Sharir, Cutting circles into pseudo-segments and improved bounds for incidences, *Discrete Comput. Geom.* 28: 475–490 (2002).

[14] P. Brass and Ch. Knauer, On counting point-hyperplane incidences, *Comput. Geom. Theory Appls.* 25: 13–20 (2003).

[15] P. Brass, W. Moser and J. Pach, *Research Problems in Discrete Geometry,* to appear.

[16] B. Chazelle and J. Friedman, A deterministic view of random sampling and its use in geometry, *Combinatorica* 10: 229–249 (1990).

[17] F.R.K. Chung, The number of different distances determined by *n* points in the plane. *J. Combin. Theory Ser.* A 36: 342–354 (1984).

[18] F.R.K. Chung, E. Szemerédi, and W.T. Trotter, The number of different distances determined by a set of points in the Euclidean plane, *Discrete Comput. Geom.* 7: 1–11 (1992).

[19] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir and E. Welzl, Combinatorial complexity bounds for arrangements of curves and spheres, *Discrete Comput. Geom.* 5: 99–160 (1990).

[20] K. Clarkson and P. Shor, Applications of random sampling in computational geometry II, *Discrete Comput. Geom.* 4: 387–421 (1989).

[21] H. Edelsbrunner, *Algorithms in Combinatorial Geometry,* Springer-Verlag, Heidelberg (1987).

[22] H. Edelsbrunner, L. Guibas and M. Sharir, The complexity and construction of many faces in arrangements of lines and of segments, *Discrete Comput. Geom.* 5: 161–196 (1990).

[23] H. Edelsbrunner, L. Guibas and M. Sharir, The complexity of many cells in arrangements of planes and related problems, *Discrete Comput. Geom.* 5: 197–216 (1990).

[24] H. Edelsbrunner and M. Sharir, A hyperplane incidence problem with applications to counting distances, in *Applied Geometry and Discrete Mathematics; The Victor Klee's Festschrift* (P. Gritzman and B. Sturmfels, eds.), AMS Press, Providence, RI, (1991) pp. 253–263.

[25] H. Edelsbrunner and E. Welzl, On the maximal number of edges of many faces in an arrangement, *J. Combin. Theory,* Ser. A 41: 159–166 (1986).

[26] G. Elekes, Sums versus products in algebra, number theory and Erdős geometry, Manuscript, (2001).

[27] P. Erdős, On sets of distances of *n* points, *Amer. Math. Monthly* 53: 248–250 (1946).

[28] P. Erdős, D. Hickerson and J. Pach, A problem of Leo Moser about repeated distances on the sphere, *Amer. Math. Monthly* 96: 569–575 (1989).

[29] P. Erdős and G. Purdy, Some extremal problems in geometry III, *Proc. 6th South-Eastern Conf. Combinatorics, Graph Theory, and Comput.,* (1975) pp. 291–308.

[30] P. Erdős and G. Purdy, Some extremal problems in geometry IV, *Proc. 7th South-Eastern Conf. Combinatorics, Graph Theory, and Comput.,* (1976) pp. 307–322.

[31] J. Erickson, New lower bounds for Hopcroft's problem, *Discrete Comput. Geom.* 16: 389–418 (1996).

[32] J. E. Goodman, Proof of a conjecture of Burr, Grünbaum and Sloane, *Discrete Math.,* 32: 27–35 (1980).

[33] N. H. Katz and G. Tardos, A new entropy inequality for the Erdős distance problem, in *Towards a Theory of Geometric Graphs,* J. Pach, Ed., *Contemporary Math.,* Vol. 342, Amer. Math. Soc., Providence, RI, (2004) pp. 119–126.

[34] F. T. Leighton, *Complexity Issues in VLSI,* MIT Press, Cambridge, MA, (1983).

[35] L. Lovász, On the ratio of optimal integral and fractional covers, *Discrete Math.* 13: 383–390 (1975).

[36] J. Matoušek, *Lectures on Discrete Geometry,* Springer Verlag, Heidelberg (2002).

[37] J. Matoušek, Range searching with efficient hierarchical cuttings, *Discrete Comput. Geom.* 10: 157–182 (1993).

[38] L. Moser, On the different distances determined by *n* points, *Amer. Math. Monthly* 59: 85–91 (1952).

[39] J. Pach, Finite point configurations, in *Handbook of Discrete and Computational Geometry,* 2nd Edition (J. O'Rourke and J. Goodman, Eds.), CRC Press, Boca Raton (2004) pp. 3–24.

[40] J. Pach and P.K. Agarwal, *Combinatorial Geometry,* Wiley Interscience, New York, (1995).

[41] J. Pach and M. Sharir, Repeated angles in the plane and related problems, *J. Combin. Theory, Ser.* A 59: 12–22 (1992).

[42] J. Pach and M. Sharir, On the number of incidences between points and curves. *Combinatorics, Probability and Computing* 7: 121–127 (1998).

[43] J. Pach and G. Tardos, Isosceles triangles determined by a planar point set, *Graphs and Combinatorics* 18: 769–779 (2002).

[44] J. Pach and G. Tóth, Graphs drawn with few crossings per edge, *Combinatorica* 17: 427–439 (1997).

[45] M. Sharir, On joints in arrangements of lines in space and related problems, *J. Combin. Theory, Ser.* A 67: 89–99 (1994).

[46] M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications,* Cambridge University Press, Cambridge-New York-Melbourne, (1995).

[47] M. Sharir and E. Welzl, Point-line incidences in space, *Combinatorics, Probability and Computing* 13: 203–220 (2004).

[48] J. Solymosi and Cs. Tóth, Distinct distances in the plane, *Discrete Comput. Geom.* 25: 629–634 (2001).

[49] J. Solymosi and V. Vu, Distinct distance in high-dimensional homogeneous sets, in *Towards a Theory of Geometric Graphs,* J. Pach, Ed., *Contemporary Math.,* Vol. 342, Amer. Math. Soc., Providence, RI, (2004) pp. 259–268.

[50] J. Spencer, E. Szemerédi and W.T. Trotter, Unit distances in the Euclidean plane, In: *Graph Theory and Combinatorics* (B. Bollobás, ed.), Academic Press, New York, (1984) pp. 293–303.

[51] L. Székely, Crossing numbers and hard Erdős problems in discrete geometry, *Combinatorics, Probability and Computing* 6: 353–358 (1997).

[52] E. Szemerédi and W.T. Trotter, Extremal problems in discrete geometry, *Combinatorica* 3: 381–392 (1983).

[53] H. Tamaki and T. Tokuyama, How to cut pseudo-parabolas into segments, *Discrete Comput. Geom.* 19: 265–290 (1998).

[54] G. Tardos, On distinct sums and distinct distances, *Advances in Mathematics,* to appear.